OULU UNIVERSITY OF
APPLIED SCIENCES

Mani S. Bhowmik

ADAPTING BASE STATION OPERABILITY SOFTWARE INTO LINUX AND SYMMETRIC MULTIPROCESSOR ARCHITECTURE

ADAPTING BASE STATION OPERABILITY SOFTWARE INTO LINUX AND SYMMETRIC MULTIPROCESSOR ARCHITECTURE

Mani S. Bhowmik
Master's thesis
Spring 2011
Degree Program in Information Technology
(Master of Engineering)
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Master of Engineering

Author: Mani S. Bhowmik
Title of Master's thesis: ADAPTING BASE STATION OPERABILITY SOFTWARE INTO LINUX AND SYMMETRIC MULTIPROCESSOR ARCHITECTURE
Supervisor(s): Dr. Kari Laitinen, Mr. Jussi Leppanen (MSc)
Term and year of completion: Spring 2011 Number of pages: 58 + 3 appendices

---

Operation and maintenance (O&M) is an application domain in the Base Transceiver Station (BTS) system. At Nokia Siemens Networks (NSN), BTS O&M Software (SW) is developed using IBM Rational Rhapsody (C++). BTS O&M SW supports the multiprocessing architecture to some extent and is mostly designed for the OSE real-time operating system that allows the best effort scheduling to be used. BTS O&M SW is not hard real time software that has to be immediately responsive to triggers, but is expected to behave deterministically in any suitable platform. The software consists of several functional subsystems with active and reactive object(s) which are executed as processes. Rhapsody Unified Modeling Language (UML) event based communication between those active and reactive objects keep BTS O&M functionality alive. BTS O&M SW mainly follows the priority based scheduling as offered by Operating System Embedded (OSE). A scheduling problem was observed when O&M SW had been ported into the Linux operating system. Unlike OSE, Linux provides a fairness-based heuristic scheduling which emphasizes the synchronizing or timing problem in existing O&M SW. This led to the birth of this thesis "*Adapting Base Station Operability Software into Linux and Symmetric Multiprocessor Architecture*".

The main purpose of this thesis is to study the synchronization problems discovered during the porting work and make the BTS O&M SW independent of scheduling, so that it can run on OSE and Linux as well as other similar symmetric multiprocessor architecture. Addressing the performance problems of O&M SW in Linux and OSE are also part of this thesis. The work includes problem study, searching for solutions, implementation and/or recommending the best solution for BTS O&M SW.

The work is done as refactoring of BTS O&M code. It is divided into development time and runtime refactoring. While the development time refactoring physically divides the code into well distinguished domains and interfaces; the runtime refactoring takes care of the scheduling, synchronization and performance problems. Runtime refactoring is the primary focus area in this work.

Keywords: base station, operability SW, O&M, Linux, OSE, scheduling, synchronization, symmetric multiprocessor, multiprocessing

PREFACE

This thesis represents the culmination of work and learning that have been carried out over the last fourteen months. In September 2009, I was assigned this thesis by Nokia Siemens Networks (NSN) as a prerequisite for my admission to Master's degree course. The actual work started in February 2010. This work has been carried out at NSN facility located in Oulu, Finland.

I would like to express my sincere thanks to NSN; Mr. Jussi Leppanen for carefully explaining the architectural details of the work and for providing me with the opportunity to be part of his refactoring dreams; Mr. Rauno Pirkola for his technical guidance at all times during the work; Mr. Jari-Pekka Tuikka for sitting by my side in the same lab corner for the last five months and suffering the similar agony of experimenting on code to prove the concepts and finally releasing few of the derived concepts; and, of course, Ms. Jaana Linna for her line management support and trust.

My sincere thanks goes to Dr Kari Laitinen for his humble, yet truly admirable ability to help writing this work in better words.

Last, but by no means least, I thank my parents and my family, specially my loving wife, Pooja, and daughter, Anusha, for their constant support and wit; and patiently accepting few of my outbursts with soothing smiles.

Oulu, Finland, May 2011

Mani S. Bhowmik

LIST OF FIGURES

CONTENTS

TERMS AND ABBREVIATION

Term	Description
BTS	Base Transceiver Station
BTS O&M	The term is used to represent BTS Operability SW in this thesis to simplify presentation. In NSN terminology, BTS O&M SW consists of operability SW and other SW used to maintain the status quo of the BTS.
CPU	Central Processing Unit
ENEA	Enea (www.enea.com) is a global software and services company focused on solutions for communication-driven products.
Event	Events provide asynchronous communication between reactive objects or tasks. Events can trigger transitions in statecharts.
I/O	Input/Output
IEEE	Institute of Electrical and Electronics Engineers (www.ieee.org). It is a “professional association dedicated to advancing technological innovation and excellence for the benefit of humanity.”
Node	A node provides a set of processing, storage and communication functions. A node hosts several logical units and have multiple CPUs.
O&M SW	Operation and Maintenance application software
OSE	Operating System Embedded, developed and distributed by ENEA
POSIX	Portable Operating System Interface. POSIX is a registered trademark of the IEEE.
Process	A process, in general, is a piece of program code that owns a virtual memory address and has a state defined by register and memory values. The term process carries different meanings in OSE and Linux architectures (APPENDIX 1).
Rhapsody	A UML based SW development tool for embedded and real-time systems. The tool was originally developed by I-Logix. Then the tool

	was owned by Telelogic and is presently owned and maintained by IBM
SW	Software
Statechart	Statecharts define the behavior of objects, including the various states that an object can enter into over its lifetime and the messages or events that cause it to transition from one state to another.
SysML	System Engineering Modelling Language
Task	A set of data dependent processes
Thread	Threads, in general, are execution context of a program. A set of threads constitute a process. A Rhapsody thread is mapped into an OSE-process or a Linux-thread (APPENDIX 1).
UML	Unified Modelling Language is a standardized general purpose modelling language in the field of SW engineering
UML call event	A UML call event is an event that represents the receipt of a request to invoke an operation. A transition with a call event initiates when the called operation is invoked.

1 INTRODUCTION

Scheduling is the practice of deciding how to commit resources between different processes and the way processes are assigned to run on the available Central Processing Units (CPUs). Scheduling is very important for BTS O&M SW. The software requires a correct-order execution of its several processes. A suitable and correct scheduling policy ensures synchronization between the tasks and secures a steady behavior of BTS. BTS O&M SW used to run on top of the Operating System Embedded (OSE) real-time operating system. The OSE provides priority-based scheduling and this guarantees that the most critical threads in the system can run immediately in response to a triggering message. Each OSE process runs program code in parallel (parallel processing) with other OSE processes within a CPU.

Parallel processing is the execution of program instructions by dividing them among multiple processors with the objective of running a program in less time. Multiprocessing is parallel processing, where two or more processors share the tasks to be done. Earlier multiprocessing systems were based on the master/slave configuration, where the slave performed the task assigned by the master, keeping the slave idle for most of the time. In a symmetric multiprocessing (SMP) system, multiple processors are equally responsible for executing a program. In a symmetric multiprocessor system, each of the processors share the same operating system and I/O bus and can either share the same memory or have their own memory space.

Real time Linux supports symmetric multiprocessing. Its scheduling is based on the time-sharing technique. The real time Linux scheduler keeps track of the processes and adjusts their priorities periodically; in this way, a process that has not used the CPU for long time is promoted by increasing its priority, while the priority is decreased of the process that has been using CPUs for a long time. Thus, it is not possible for the SW designer to specify an absolute highest priority process.

BTS O&M SW is designed to execute a very large number of processes, each with a predefined priority. When the BTS O&M SW runs on OSE, the set priorities of the processes provide an instrument to OSE, to schedule them in harmony achieving pure synchronization between the processes. Since the OS decides the running sequence of the processes based on pre-defined priorities, it sometimes fails due to starvation when a lower priority process is denied

use of the CPU time as a higher priority process is still using the CPU. Such starvation problems are solved by fine tuning the priority of the process to meet the increasing size and functionality of the process. When this OSE based BTS O&M SW was ported into the Linux real time operating system, synchronization problems popped up. In order to get rid of the priority based O&M SW, equal priority or zero priority for all process based scheduling is chosen for the Linux real-time OS. Such a Linux scheduling policy ignores the predefined process priority and executes the processes in a dynamic order; which is mostly out of order execution of BTS O&M SW. In addition, the inherent requirement of every software application is to achieve a higher level of performance. The performance requirement varies from system to system. An important software application such as BTS O&M is expected to demonstrate a high level of performance, especially during start ups and recovery actions.

2 THEORETICAL BACKGROUND

This thesis requires some theoretical understanding of the concerned real time operating systems and symmetric multiprocessor architecture. These theoretical presentations help to understand the problems and their recommended or provided solutions. Rhapsody - the tool used for O&M SW development; OSE and Linux - the concerned operating systems and symmetric multiprocessor architecture to which the BTS O&M SW is executed, are discussed in the following sections.

2.1 Symmetric Multiprocessor (SMP) Architecture

A multiprocessor system supports more than one central processing unit (CPU) and is able to allocate tasks between them. In a multiprocessing system, the tasks are distributed equally to the CPUs, or some of the CPU may be reserved for a special purpose that can execute a limited set of instructions. When all CPUs of a multiprocessor system are treated equally, it is called a symmetric multiprocessor system. According to Flynn's taxonomy (Flynn, 1972, 2009), Single Instruction, Multiple Data (SIMD) is a multiprocessing environment where the processors are used to execute a single sequence of instructions in multiple contexts. SIMD is often used in vector processing. In Multiple Instructions, Single Data (MISD) environment, multiple sequences of instructions are executed in a single context used for redundancy in fail-safe systems; and in Multiple Instructions, Multiple Data (MIMD) environment, multiple sequences of instructions are executed in multiple contexts.

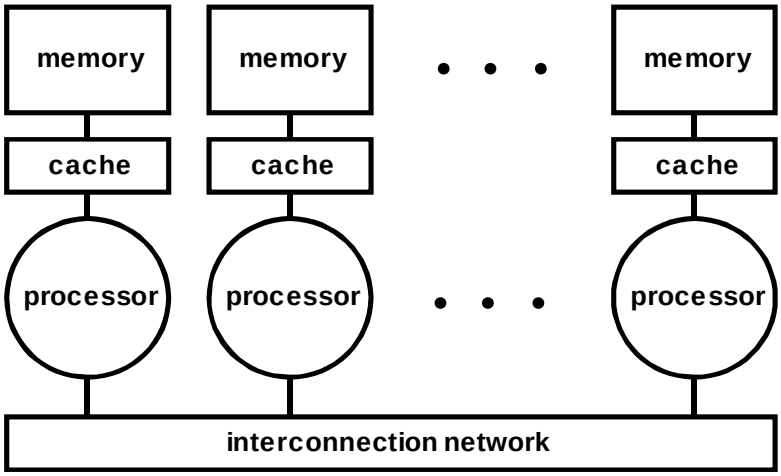

Figure 2-1 Message passing architecture (Thornley, 1997)

A multiprocessor message-passing architecture can have a separate address space for each processor and the processors communicate via messaging (Figure 2-1). In a memory-sharing architecture, all processors share a single address space and communicate by memory read and write (Figure 2-2). What makes multiprocessor architecture symmetric is the equal closeness of processors and the memory. In case of multi-core processors, the SMP architecture applies to the cores, treating them as separate processors. Processors, in SMP architecture, are interconnected by a shared bus, cross-bar switches or on-chip mesh networks.

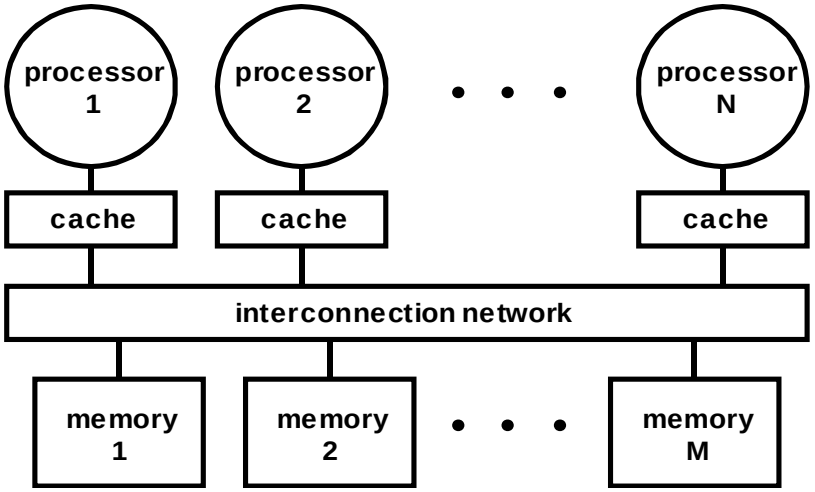

Figure 2-2 Shared memory architecture (Thornley, 1997)

Interconnect is a finite resource (Herlihy, 2008) in terms of bandwidth and is shared by multiple processors. The finite bandwidth of interconnect and power consumption during communication processes cause a bottleneck in the scalability. Processors can be held up if others are consuming too much of the interconnect network. Mesh architectures can provide nearly linear scalability but multi-task programming is very difficult for such an environment.

The symmetric multiprocessing systems require different programming methods to achieve the maximum performance. SMP has many uses in science, industry and business SW if it is designed for multithreaded or multitasking processing. It is still to be noted that the programs running on the SMP systems may experience a performance increase even if those have been written for single processor systems. This is due to the kernel selection of an idle processor for the execution of the process that is suspended by a hardware interrupt. In some applications, particularly the compilers and some of the distributed computing projects, the performance can be increased nearly by a factor of the number of additional processors.

In situations where more than one program executes at the same time, an SMP system yields considerably better performance because different programs can run on different CPUs simultaneously. In cases where an SMP environment processes many tasks, administrators often experience a loss of hardware efficiency. The software programs have been developed to schedule tasks in a way that the processor utilization reaches its maximum potential. A good software packages can achieve this maximum potential by scheduling each CPU separately, as well as being able to integrate multiple SMP machines and clusters. A serialized access to the memory and cache coherency problems causes the performance to lag slightly behind the number of additional processors in the system.

The BTS O&M application SW is divided into several sub-applications. During the runtime, each of the sub-applications is divided further into several processes which run in parallel. Those processes are distributed between processors in one or several nodes. The multiprocessing interpretations available for the BTS O&M SW are: Single Node Single Processor (SNSP), Single Node Multiple Processor (SNMP) and Multiple Node Multiple Processor (MNMP). The distribution mechanism is defined in the runtime architectural design of the sub-applications. Each of these processes communicates with each other locally using the Rhapsody UML events, when they are in the same processor or via messages with the help of the distributed framework when the processes are distributed in several processors.

2.2 Operating System Embedded (OSE)

ENEA OSE is a real-time operating system and supports the multiprocessor architecture via a high-level message passing programming model (Figure 2-1). Thus, it is easy to break down a complex program into simpler concurrent processes which communicate via high speed direct messages. The OSE kernel provides basic services such as pre-emptive priority-based scheduling, and direct and asynchronous message passing for the inter-task communication and synchronization. A fault tolerant distributed system can be built on OSE. A good program made on OSE can enjoy a deterministic real time behavior. OSE provides a powerful API with high level of abstraction, enabling programmers to code the bulk of their application with only eight system calls. This versatile API, together with the high-level messaging protocol of OSE, reduces application size and complexity, making programs easier to maintain, read and understand. OSE Inter Process Communication (IPC) services extend the benefits of message passing to OSE applications distributed across multiple processors. (ENEA, 2011)

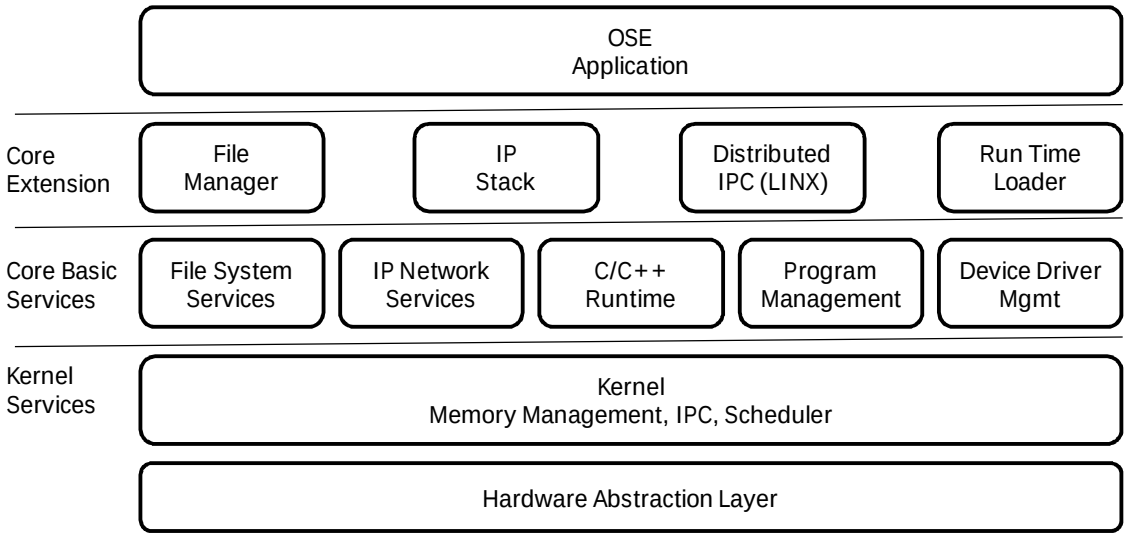

Figure 2-3 Modular and layered architecture of OSE (ENEA, 2008)

The asynchronous message passing architecture and programming model of OSE facilitates a modular system design and reduces the complexity and lowers the maintenance costs. End-product reliability, availability and robustness are increased by its built-in supervision, resource failure detection and error handling mechanism. The distributed design is simplified by transparent communications between the processes residing on multiple CPUs. It also makes the

systems easier to configure, scale and upgrade. Memory protection increases the robustness and security of the program and provides simplified debugging techniques. The pre-emptive, deterministic real-time response of OSE is suitable for high-availability and mission-critical applications.

2.3 Real time Linux

Real time Linux is an operating system that differs from the standard Linux. The BTS O&M SW being a semi-hard or firm real time (in between soft and hard) system makes it fundamentally suitable to execute on the Linux real time environment. Real-time Linux can be considered as a viable candidate for real-time applications as it has gained its maturity in recent years. Several real time applications on real time Linux have demonstrated a successful real-time behavior. Different research groups have proved the stability of real time Linux which has given it a boost to be commercially available as a product. Open source software ensures the future maintainability and extensibility of software systems. Real-time versions of Linux offer important advantages to control-engineers in providing an open source operating system that rivals the performance of the proprietary real time kernels. The Linux kernel has been constantly under modifications which have resulted in reduction of both the interrupt latency (the time delay from an interrupt to the start of the processing that interrupts) and jitter (variations in the timing of periodic events) to the microsecond range, allowing a faster response to external events and higher resolution timing. Over time, Linux has become a very suitable choice for embedded system development (Aeolean Inc, 2002).

There are some basic differences between the standard and real time Linux. Unlike in the standard Linux, in real time Linux architecture the interrupt processing is divided into two sections, as top-half and bottom-half tasks. The bottom-half task is the interrupt handler that reads data from the physical device into a memory buffer. The top-half task reads from the memory buffer and passes the data to a kernel accessible buffer. This ensures an improved latency and immediate service to subsequent interrupts when the previous one is still under process. There are several implementation styles to make a standard Linux a real time Linux.

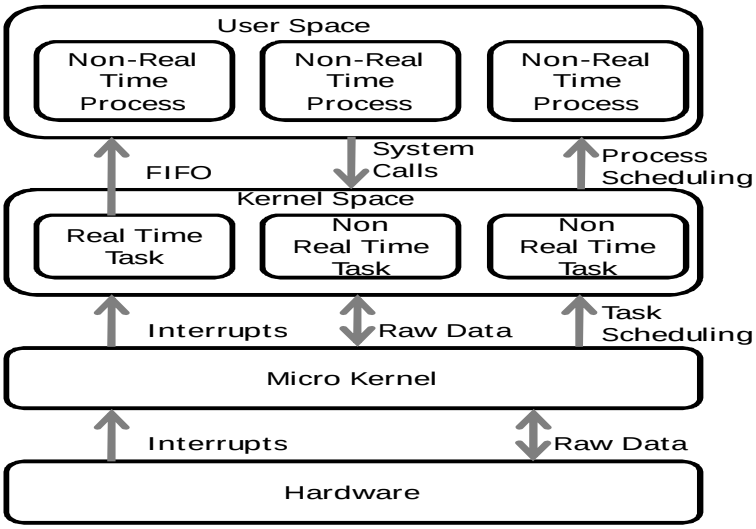

Figure 2-4 Linux real time micro kernel architecture (Aeolean Inc, 2002)

In the Micro Kernel style, a second kernel serves as an interface between the hardware and standard kernel. This compact code module, or micro kernel, handles the execution of the real time operations, while the standard kernel takes care of the standard tasks in the background. The micro kernel prevents the standard kernel to pre-empt any interrupt processing in the micro kernel and schedules the real-time tasks with the highest possible priority to minimize the task latency. Figure 2-4 illustrates the micro kernel architecture.

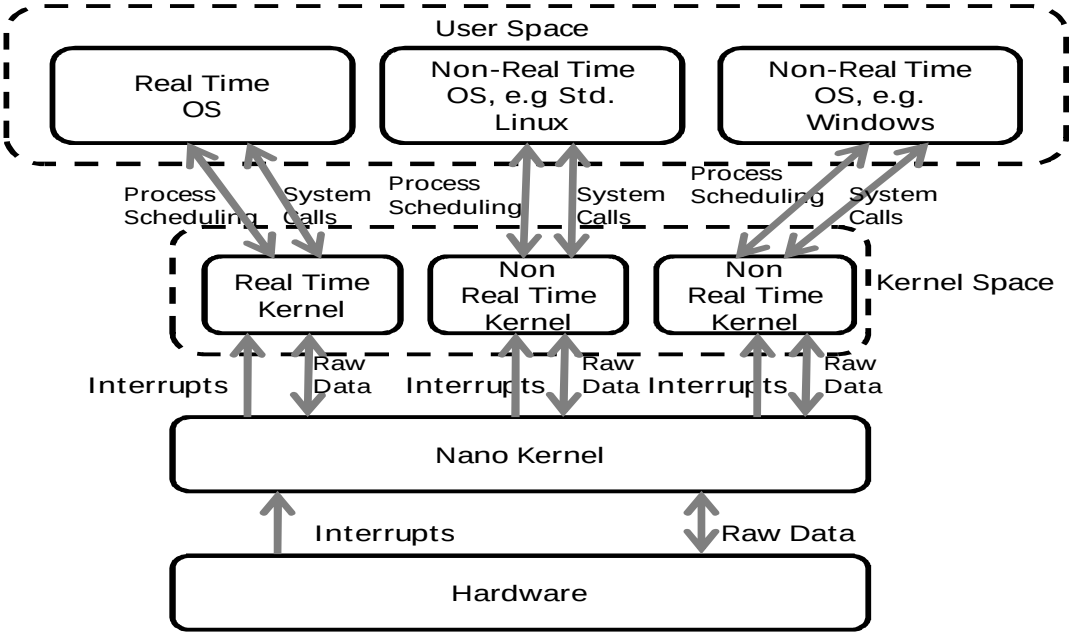

Figure 2-5 Linux real time nano kernel architecture (Aeolean Inc, 2002)

The second style is the Nano Kernel (Figure *2-5*) in which the philosophy is similar to the concept in micro kernel implementation, but differs in the design approach. The Nano Kernel design approach makes it possible to run many operating systems on top of the nano kernel.

The Portable Operating System Interface (POSIX) Real Time Extension style is to modify the standard kernel directly according to the IEEE 1003.1d standard. There is no extra kernel in this architecture.

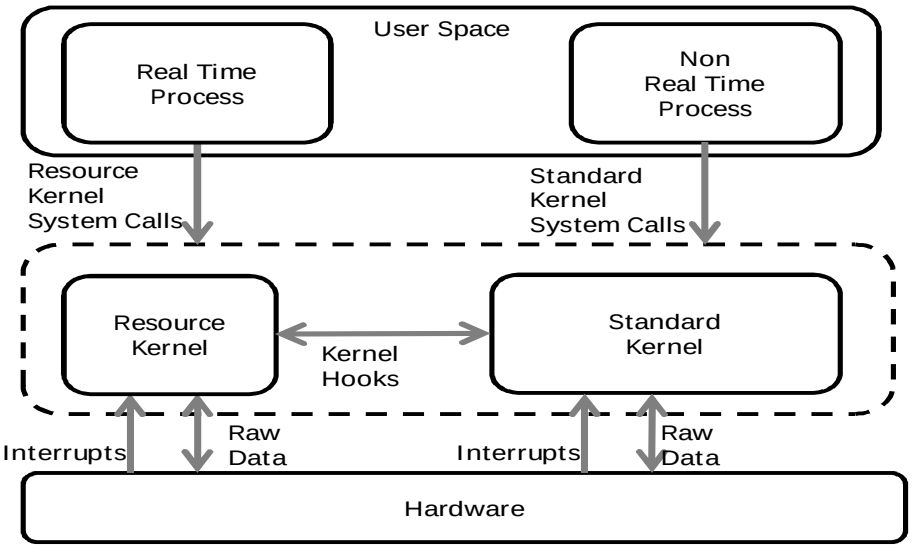

Figure 2-6 Real time Linux resource kernel extension architecture (Aeolean Inc, 2002)

Resource Kernel Extension (Figure 2-6) is an example of such an approach. In this approach, a resource kernel is designed as a compact gateway for external interrupts. Apart from the preemption of external interrupts, the resource kernel also guarantees finite resources, such as memory, CPU cycles, network, and file system transfer bandwidth for the user-space applications.

2.4 UML and Rhapsody

IBM Rational Rhapsody is a Unified Modelling Language (UML) tool that provides graphical notation associated to the UML, especially for the software system that is built using the object oriented methodology. Rational Rhapsody provides code generation from model, diagramming - creating and editing UML diagrams as well as round trip engineering – code generation from

model and model generation from code and reverse engineering – deriving model diagrams from the source code.

Rational Rhapsody Developer is used to generate a full behavioral code in C, C++, Java or Ada for real time operating systems. It provides an environment that enables an early validation of the behavior of the software by using rapid prototyping, visual debugging and model execution. Rational Rhapsody Designer is used by systems engineers to simulate early requirements. It helps the engineer to validate the architecture and behaviour of the system. For real-time and embedded software development, the Rational Rhapsody Architect for software provides an UML and System Engineering Modelling Language (SysML) based software development environment. Embedded software developers can leverage an integrated software development environment for C, C++ or Java code that automatically helps to improve application consistency through UML based modelling, maintaining the consistency of architecture, design, code and documentation. Similarly, the Rational Rhapsody Architect for Systems Engineers helps systems engineers to manage the complexity of the developed products and specify cohesive architectures and designs.

The BTS O&M SW is developed using Rhapsody C++. This means that the model diagram is converted to C++ source code. Rhapsody generates the code in an OS-independent fashion. This is achieved with the use of a configurable application framework called the Object Execution Framework (OXF). The OXF (Figure 2-7) is provided with Rhapsody in a model form, and is generally built as a library that is linked with the Rhapsody generated application code. OXF provides critical real-time services such as threading, synchronous and asynchronous messaging between objects, resource protection and timeouts. The O&M SW design follows the object oriented style that contains packages and classes for data hiding, inheritance, interfaces and polymorphism; while the behavioral aspect of the SW is achieved by UML state charts.

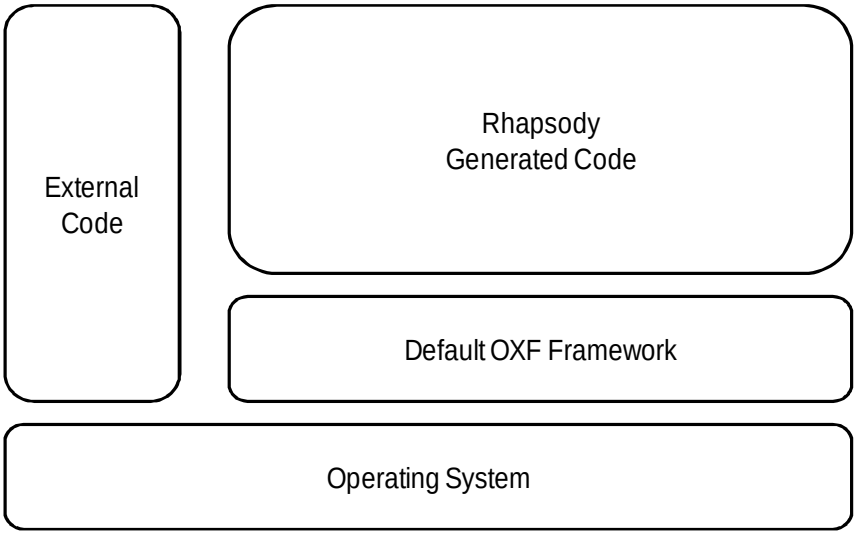

Figure 2-7 Location of Rhapsody OXF model

In Rhapsody terminology, a class is reactive if it has a statechart or consumes an event or is a composite. A reactive object is an object that receives and processes an event. Reactive objects are either active or sequential. The Rhapsody OXF framework creates a thread for each reactive object. In the OSE execution environment, Rhapsody threads are mapped to OSE processes, and in the Linux execution environment the Rhapsody threads are mapped to Linux threads (APPENDIX 1).

An active object is the owner of the control thread and initiates the control activity. Both active and reactive objects run on their own threads. In a multi-threaded application, Rhapsody generates active objects with the primary thread and any number of reactive objects with additional threads. By default, a sequential object with a state chart shares the thread and the event queue of their parent object, unless they are also active, in which case each of them owns their own thread. A sequential object does not initiate any control activity but can hold the data and behavior as that of an active object. The OMReactive class is the framework base class for all reactive objects (Rhapsody Help, 2010).

A Rhapsody reactive object has a public member function called startBehavior(). This operation initializes the behavioral mechanism of an object and takes the initial transitions in the state chart. The startBehavior() is called on the thread that creates the reactive object, and the default transitions are taken on the creator thread. startBehavior should manually be invoked when a reactive object is created manually (in user code); otherwise the reactive object does not respond to the events.

The message communication between the objects is done either via the synchronous interface (call and wait for return, such as a function) or the asynchronous interface (sends and continues, such as an event). An active object is created with an associated message queue and manages the asynchronous messages sent to itself or to sequential objects that are set to be executed on the active object. Asynchronous messages such as signal events and time events are queued, and then processed on the receiving thread. Synchronous messages, such as functions, simple operations and UML call events, are executed on the caller thread and bypass the message queue.

The message queue serves the active threads with the signal as the First In First Out (FIFO) mode and is protected from the concurrent access by different threads. The message queue is a buffer that helps the independent but cooperating tasks to maintain the asynchronous communication between each other (Mayer, 2005). The message queue is essential in the non-shared memory or message passing (Section 2.1) system to preserve the asynchronous behavior of the system. An event, meant for another class, is passed to the operating system message queue and the target class retrieves the event from the head of the message queue when it is ready to process it.

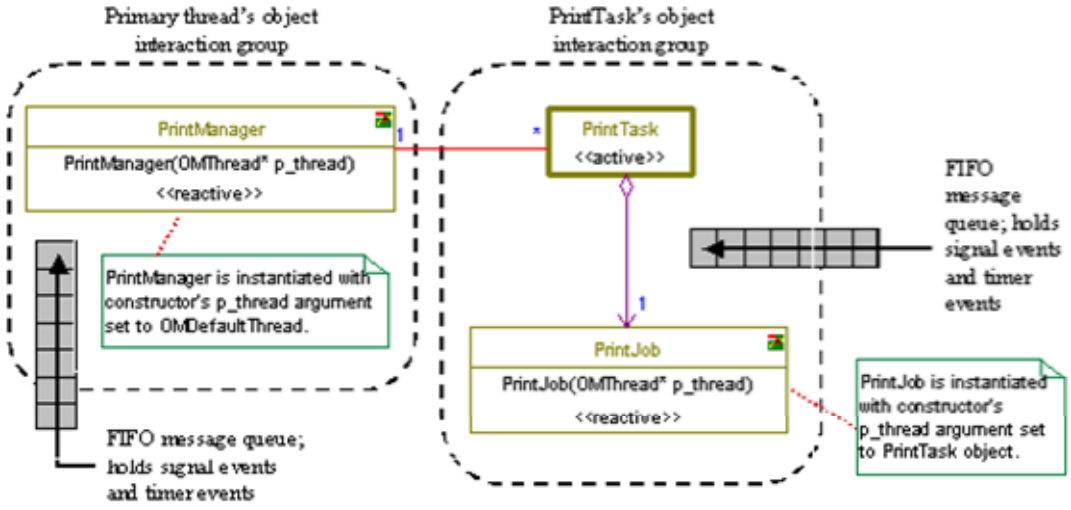

Figure 2-8 Message queue, thread of control (Mayer, 2005)

Figure 2-8 shows an example of a message queuing mechanism and of the usage of a thread of control. Since it is specified, the reactive object PrintJob runs under the control thread PrintTask, while PrintManager is being a reactive object runs on the default thread. In both

cases the message queue holds the message signal (UML events and timers) until the previous signal is consumed.

The processes in a non-shared memory system must be linked to each other as the message queue is attached to the link that allows the sender and receiver of the message to continue with their own processing activities independently. Rhapsody provides a unidirectional or bi-directional association between the classes. The message flow is always towards the link, i.e. in the case of a unidirectional link from class A to B, A can send the message while B can receive, but vice versa is not possible. In the bi-directional link both ends can send and receive messages.

3 THESIS STATEMENT AND PURPOSE

This thesis is intended to study, specify and, possibly, implement supports for the improved synchronization into the BTS O&M SW code. The thesis also intends to examine the O&M code for potential problems and provide corrections or recommend solutions and create guidelines. Mainly, this thesis is to make the software robust for OSE, Linux and other similar symmetric multiprocessor architectures.

3.1 Research Question

The thesis is matured around the following question:

- *How to make the BTS O&M software independent of OS befitting both OSE and Linux and other symmetric multiprocessor architectures as a better performing and reliable software?*

3.2 Thesis Objective

The primary objective of this thesis is to construct the basis for the BTS O&M SW to make it an OS independent software that can run on the existing OSE and simultaneously can be ported to Linux or similar symmetric multiprocessor environments. The defined goal of the thesis is to study the possibilities to reduce the number of BTS O&M SW threads in order to increase the performance of the system and to synchronize the threads to run on both the “priority based - OSE” and “time-slice based - Linux” real-time operating systems. The secondary objective of the work is to optimize the BTS O&M SW to reinforce the original architectural model and provide a safer approach for future developments.

3.3 Reliability Requirements

In the future BTS will have more users (capacity increases) and more purposes of usage (calls and wireless broadband). Therefore, the requirements for the reliable operation and maintenance of the BTS are higher than before. A reliable BTS must have:

- Safer HW
- Safer SW (no SW bugs)
    - Faster recovery from failure
    - Fast start up
    - Less failure
- Problem restriction into a small area. This means that if some part of the BTS is faulty, the whole BTS does not need to reset
- More flexible SW is needed. For example, it must be possible to adapt the situation if one part of the BTS starts up when another part is up and running

The Linux real time OS will be used in the BTS because it provides a better tool/driver support and is cost-effective. The same SW must work on OSE and Linux and problems caused by different scheduling mechanisms must be solved. A symmetric multi-core processor will be used in the BTS. Thus, the problems caused by parallelism must be solved and SW must be optimized for parallel execution.

4 BTS OPERABILITY SOFTWARE

Base Transceiver Station (BTS) or Cell Site is used to facilitate wireless communication between User Equipment (UE) and wireless communication network. BTS functionality can be divided broadly into four planes: Management, Radio Network Control, Transport Network and Radio Network User Planes. As illustrated in Figure 4-1, BTS Operability (O&M) SW is the Management Plane application software that communicates with lower layers for detection and configuration of the HW units. O&M SW provides configuration information to telecom service for the creation and maintenance of the cells. Runtime health-check of the modules and reaction to any anomalies is also part of the O&M job description. O&M SW responsibilities can be mapped to the well known FCAPS model. The BTS O&M SW is responsible for Fault Management, Configuration Management, Administration, Performance Management and Security Management.

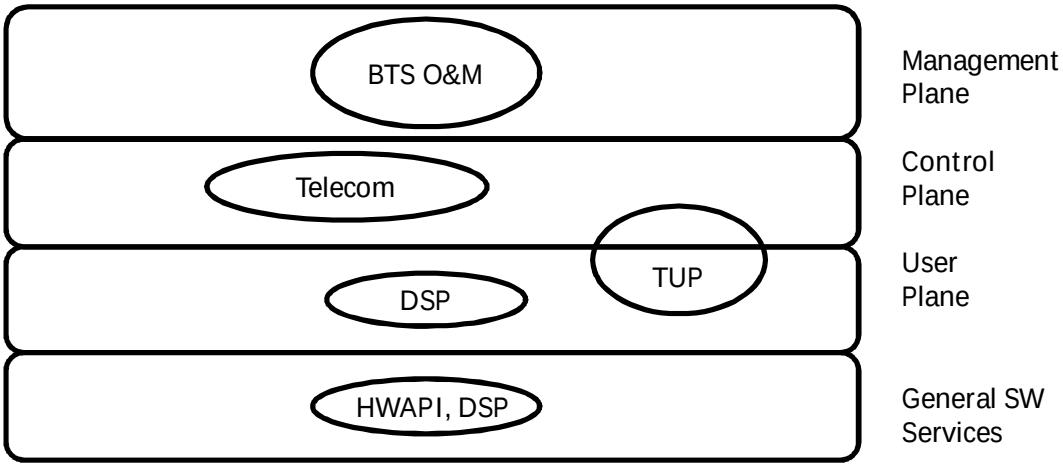

Figure 4-1 BTS functional planes: O&M position

Fault Management in the BTS O&M SW is responsible for fault monitoring and reporting. Fault Management detects and isolates the faults raised by an inappropriate operation of a HW or a logical unit. The isolation is necessary to take any recovery actions on the unit and reconfigure the unit to make it fully operational. The faults are also logged for analysis purposes in order to design preventive actions.

Configuration Management is responsible for the detection and configuration of the HW units in BTS. Configuration Management works in harmony with Fault Management for recovery

actions and reconfiguration of the faulty units. Configuration Management is also responsible for Network Provisioning by reading the user-defined configuration data and distributing them to the appropriate applications. Configuration Management is responsible for the database creation and handles the configuration changes in BTS.

Administration is the part where O&M SW tracks the services and usage of resources. Temperature Management of the HW, Testability, SW Management, Reset Management, Time Synchronization and License Management are part of administration.

Temperature Management or Climate Control is to maintain a certain temperature of the BTS cabinet and modules to avoid HW burn and malfunction; Testability SW covers all automatic testing and diagnostic problems in BTS. SW Management is responsible for downloading, uploading, installing and activation of different runtime SW for all BTS units. Time and synchronization management is responsible for delivering system time, tuning the system clock and clock burst or pulse counting. License management is responsible for runtime variability management and license based feature management of the BTS.

Performance management involves the periodic collection of quality of service metrics or performance counters which characterize the performance of BTS resources. Finally, Security Management or authentication services are used to authenticate the management user, and node management carries the responsibility for authenticating BTS into the radio access network (RAN).

4.1 OSE scheduling in BTS Operability SW

ENEA OSE scheduler supports the priority based FIFO scheduler policy. OSE manages the application process execution through the priority-based pre-emptive scheduling. CPUs serve the process in the same order as they are ready to run. In addition, since the processes are pre-empted by the predefined priorities, a higher priority process gets the CPU time rather than the process with a lower priority. The governing principle is that the highest priority process ready to run should always be the process that is running.

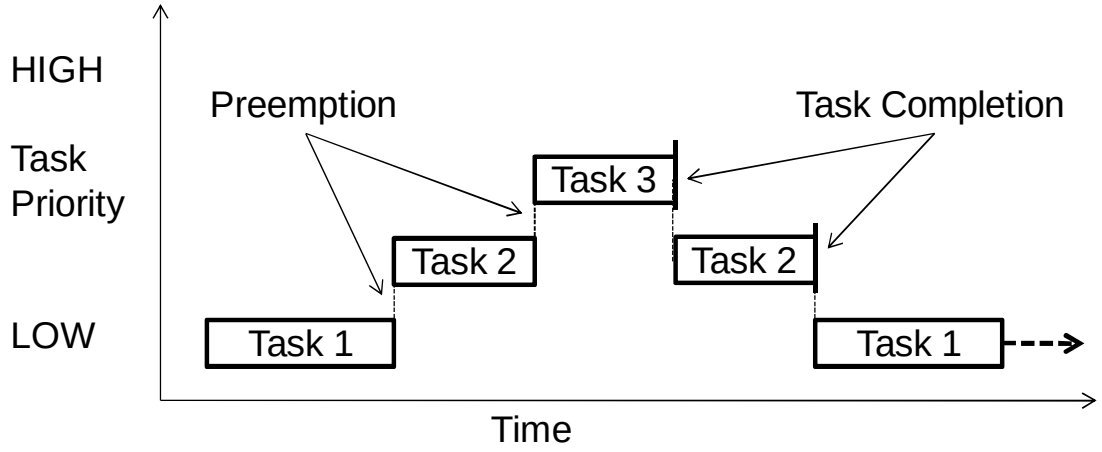

Figure 4-2 Preemptive priority-based scheduling

As described in Section 4, O&M SW functionality is divided into several domains. Each domain has several subsystems. Each of these subsystems has a main class which is the entry point for the subsystem. Each of these subsystems runs in their own Rhapsody thread (Section 2.4) and is mapped into an OSE process. All these OSE processes or Rhapsody threads are executed according to given static priorities. When O&M execution starts, OSE kernel runs the threads in a preemptive manner. Whenever a thread with highest priority is ready to run, the kernel executes the thread by suspending the execution of the lower priority thread. Thus, each of the OSE processes gets the required CPU time to complete their tasks. The priority-based scheduler does not modify the process priority dynamically, and therefore it is responsibility of the designer to set the priority according to the execution sequence of the application.

4.2 Linux scheduling in BTS Operability SW

Linux scheduling is generally based on the time sharing technique where several processes run in time multiplexing. The CPU time is divided into slices, one for each runnable process. SCHED_OTHER (Linux Manual) is the default universal time-sharing scheduler policy used by most systems. SCHED_BATCH is intended for the "batch" style execution of processes. SCHED_FIFO and SCHED_RR are intended for special time-critical applications that need a precise control over the runnable processes which are selected for the execution. A single processor can run only one process at any given instant. If a currently running process does not stop the execution when its time slice or quantum expires, a process switch may take place. The

time sharing relies on timer interrupts and is thus transparent to the processes. The scheduling policy is also based on ranking the processes according to their dynamic priority.

All scheduling is pre-emptive. The process priorities are set either dynamically or statically. In any case, the real time priority determines the execution order and pre-emption. When the process priority is set statically, a process with a higher priority gets the attention of the kernel and the current process is pre-empted and returns into its wait list. When a static priority value, sched_priority is assigned to each process, this value can only be changed via system calls. The scheduling policy only determines the ordering within the list of the runnable processes with an equal static priority.

When the process priorities are dynamic, the scheduler keeps track of the processes and adjusts the process priorities periodically. Conceptually, the scheduler maintains a list of runnable processes for each possible _priority value in the range from 0 to 99. The processes scheduled with SCHED_OTHER or SCHED_BATCH must be assigned the static priority 0 (Linux Manual). The processes scheduled under SCHED_FIFO (First In First Out) or SCHED_RR (round robin) can have a static priority in the range 1 to 99. Each process associated with such a value tells the scheduler how appropriate it is to let the process run on a CPU. In order to determine the process that runs next, the Linux scheduler looks for the non-empty list with the highest static priority and takes the process at the head of this list. The scheduling policy determines, for each process, where it is inserted into the list of processes with an equal static priority and how it moves inside this list. Processes which are denied the use of the CPU for a long time are boosted by dynamically increasing their priority, and the processes running for a long time are set to decreased priorities.

There are three classifications of the processes: Interactive processes, which constantly interact with their actors or users; Batch processes, which do not need user interaction and run in the background; and Real-time processes, which have stringent scheduling requirements and should never be blocked by the lower-priority processes. The traditional classifications of the processes are done as I/O-bound or CPU-bound. A batch process can be either I/O-bound or CPU-bound. The real time processes are explicitly recognized as such by the scheduling algorithm in Linux. The Linux 2.6 scheduler implements a sophisticated heuristic algorithm based on past the behavior of the processes to determine if a process is batch or interactive in nature. The Linux scheduler tends to favour interactive processes over the batch process. Every real-time process is associated with a real-time priority. A scheduler always prefers a higher priority runnable process over a lower priority process.

SCHED_OTHER is a conventional time-shared model and is chosen in Linux for the BTS O&M SW to provide an equal chance for all process threads and to get rid of the OSE based priority execution architecture where a resource hungry higher priority eats up all CPU time leaving very little CPU time for the lower priority process. Thus, in the chosen Linux real-time system, the execution of the processes is determined on a time sharing basis giving all processes a fair chance to complete the functional behavior. In the chosen Linux version 2.6, the scheduler is smart not to scan all the tasks each time. Rather, a ready process is arranged into a favourable position in the current queue. The scheduler chooses the task from the queue. In addition, scheduling is done in a constant amount of time. A running process is allowed to run for a given period of time. On the expiry of the time, another process is chosen from the queue while the previous process is moved to the expired queue, and sorted according to the runtime priority. Once all the processes in the current queue are executed, the queue switch takes place and the previous expired queue becomes the current queue and vice versa. The scheduler resumes executing the processes from the new current queue again.

5 REFACTORING

This thesis is done within the refactoring project scope. Refactoring is basically aimed to improve the design of the existing code in such a way that it is easier to understand and easy to modify without breaking or changing the functional behaviour. Although the main reason behind this project was to make the O&M SW suitable for both OSE and Linux architectures, it also gave the project a golden opportunity to realize the long-term goal to create robust software and stop the decaying of the design that had been done several years ago. It is well understood that during the porting of O&M SW into Linux, that refactoring is needed to retain the shape of the original architectural design of O&M SW.

Refactoring is aimed to study, identify and provide or recommend solutions to make the software fit for both OSE and Linux real-time systems. The refactoring project is divided into development time refactoring and runtime refactoring. While the development time refactoring is aimed to delineate software into domains and precise interfaces, the runtime refactoring is aimed to sort out the synchronization problems and obtain a faster start up.

5.1 Development time refactoring

BTS O&M is very old software which was developed with Rhapsody C++ as a single Rhapsody project. Following the traditional coding style of NSN GSM BTSs, the software had been divided into several domains. Each domain had a well defined interface. Over time, several other products were supported by reusing the code where the initial development style had been manipulated to suit the needs of the project and the taste of the developers. Tight project schedules and lack of development guidelines worsened the situation and O&M SW had become very difficult to maintain and it became enormous in size. Thus, it had been very important to do the development time refactoring to realign the original development ideas of the BTS O&M SW and keep O&M distributed into well defined domains and their interfaces.

5.1.1 Independent domain based Rhapsody model

Rhapsody C++ based BTS O&M SW was developed for a BTS product approximately 10 years ago. Newer BTS products had been developed on top of the old code by reusing the existing code and introducing new sets of the product specific code. Thus, every newer BTS product development had added a new code while the old code was reused to a feasible extent. Each domain in the model provided one or more services and accordingly their interfaces were defined. However, during the incremental development, the service(s) provided by the domains were mixed up and the service interfaces lost their focus.

Thus, it has become a momentous task to realign the domains and their service interfaces. The whole BTS O&M project is divided into several domains and further separated into independent projects. A flat domain-based model, where each domain is created combining the subsystem of a similar logical functionality, is illustrated in Figure 5-1.

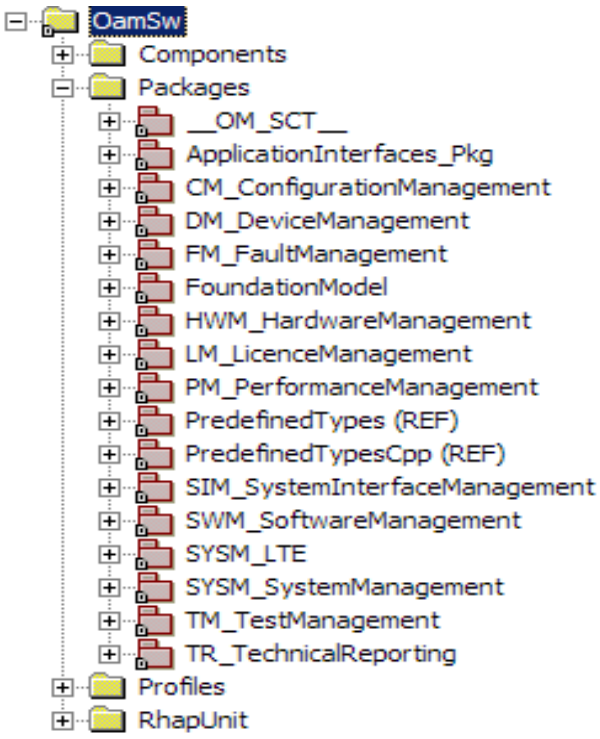

Figure 5-1 Domain based flat project model

Figure 5-2 shows the repository directory structure of the domain based split model where each of the domains is an independent Rhapsody model and can independently be worked

on. This separation of the domain is done to increase the focus on a single domain and the maintenance of the interface and functional behavior. One such independent domain is illustrated in Figure 5-3.

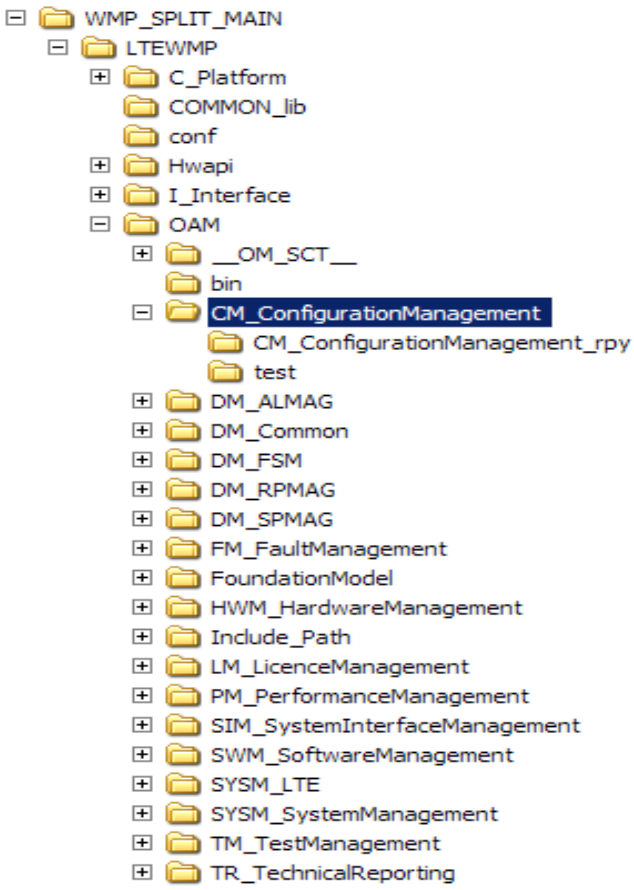

Figure 5-2 Repository directory structure

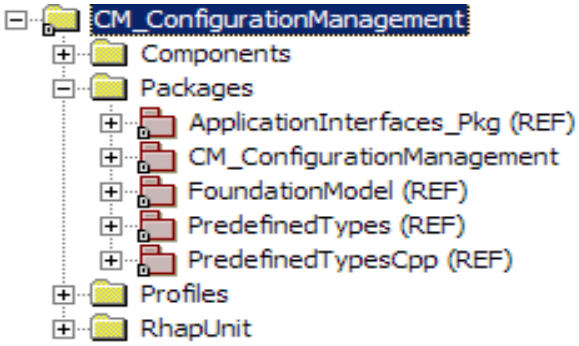

Figure 5-3 Example of domain based split model

A closer look at Figure 5-3 reveals the reference subsystems as REF. The FoundationModel is referenced with the independent model with a unified direction of the interfaces used between the subsystems. Directional interfaces are discussed further in section 5.1.3.

5.1.2 Unused code removal

BTS O&M SW has a long history of development and reuse. After each product development and release, the same code was reused to develop the next products. Thus, some of the old code became redundant. The 3G BTS has more than a decade-long history of the development of newer and smarter products. Thus, the amount of such redundant code increased with time. The product specific code was separated by using compile time pre-processor flags (#ifdef PRODUCT, #ifndef PRODUCT, #if defined PRODUCT, #if !defined PRODUCT). Because of different requirements, HW and wrong design approach, some part of the code became repetitive, only to be separated by pre processors. This led to a poor design where usually more code was required to do the same tasks. Because there was more code, it was more laborious to modify correctly. The essence of a good design is to have a piece of code once that says everything only once (Fowler, 1999).

Thus, all such redundant code was removed to enhance the readability and easier maintenance of the SW. The usage of the unused code was not only separated using compiler flags; sometimes, the unused code was put under the conditional method call that decided if the code under the condition should be executed and this decision was done in runtime. Thus, the unused code still exists in runtime binary of O&M. This is presented in Section 5.2.9 as runtime refactoring.

5.1.3 Changing bidirectional association to unidirectional

During the creation of the BTS O&M SW the subsystem classes were strongly coupled by the bi-directional association or two-way linking. The Bi-directional association might be useful when a software size is relatively small, but as the SW has grown larger, it is proved to be a costly affair to maintain the two-way links. The philosophy behind the bi-directional association between the classes is discussed in Section 2.4. The possibility of designing a simpler communication

mechanism between the classes has made the designer believe that this might be useful. But this has come at a cost of harder maintenance and more mistakes in the later phase to create and remove objects during the creation and deletion processes. This is because the bi-directional association imposes the interdependency between the classes in the development time and between the objects in the runtime. Moreover, when the classes are in different packages it also creates interdependency between the packages. Figure 5-4 exemplifies such misguided design.

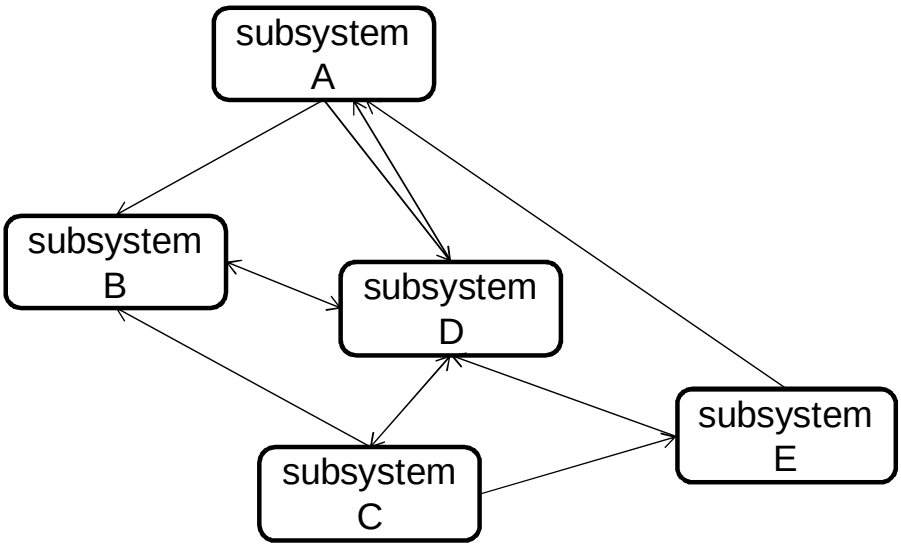

Figure 5-4 Interdependency between subsystems

One of the aims of the development time refactoring is to identify such hard-coupling and change them to unidirectional association and make a clear separation between the server and client. When needed, such bidirectional association is achieved via defined interfaces. Thus the accessor association is changed to bidirectional, but the implementation remains unidirectional. Figure 5-5 shows an example of unidirectional relation via an interface.

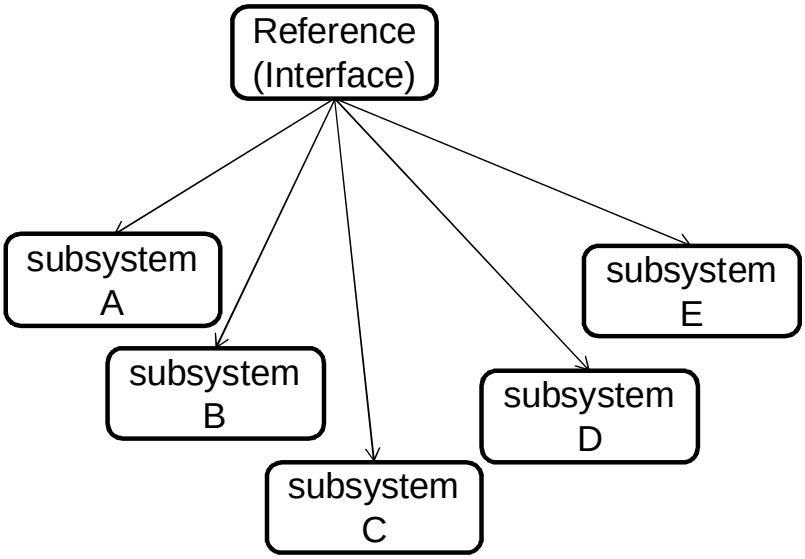

Figure 5-5 Simplified association between subsystems (Leppanen, 2010)

5.1.4 Common database

BTS SW is a combination of several system components. Among them, BTS O&M SW owns the database. Other system components communicate to the dedicated O&M application to access different sets of data. Thus, the runtime update of the database requires a continuous query and update messaging between the database, database-responsible application and database-update initiator. This style of database access slows down the BTS start up because of the extra messaging over-head.

For example, application A has a direct access to the database (D); application B asks for a required data from application A. Thus, four communications between B-A, A-D, D-A and A-B occur. If the database is available to B directly, the same information is achieved using two communication events; e.g. B-D and D-B. In first approach, if the queried data is not available for the first time, the overhead of the extra two communications increases two fold due to every subsequent attempt until success. Moreover, A has to carry an extra overhead (in addition to its own general responsibilities) of providing data to application B, whenever B approaches A for the data.

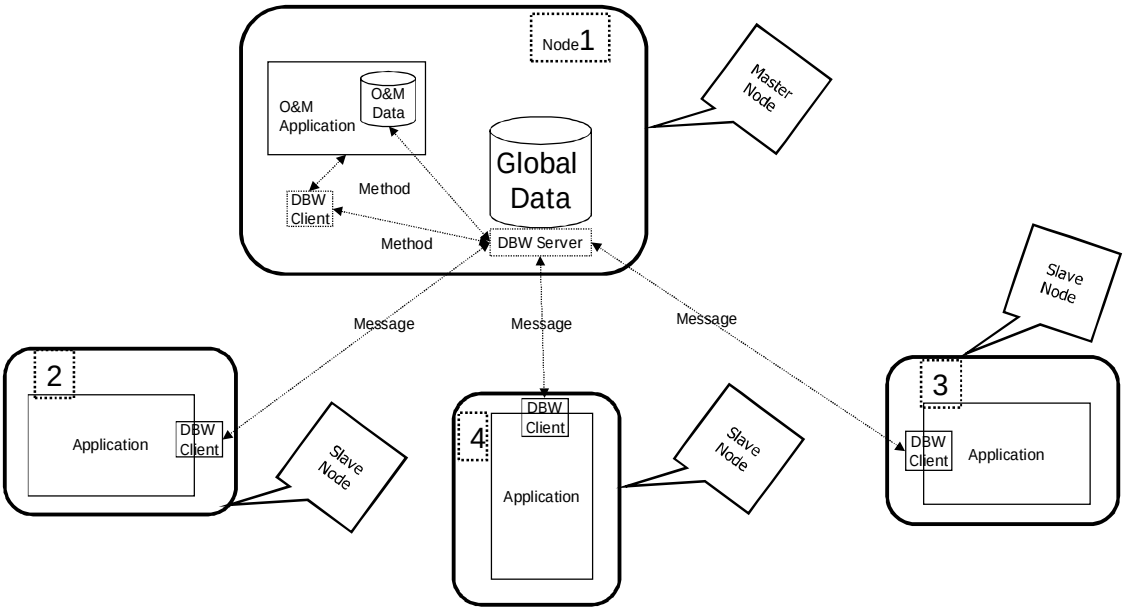

Figure 5-6 Common database, data centric design

BTS SW is a combination of several applications running in different processor spaces. These applications communicate with each other and with the master application using a predefined interface based on messaging or method calls. Most of the applications in the BTS SW depend on each other for the configuration data. If any of the applications stops executing, the dependent applications cannot continue either. This dependency can be reduced using a common database (Figure *5-6*) where all updated information is kept and provided when requested. If any application stops and restarts, the most recent data is still accessible, and thus restarting of the whole BTS might be avoided. The failing application can recover using the configured data in a shorter time and BTS becomes fully operational quicker.

If the configuration is modified in the runtime, some of the configuration changes trigger the BTS reset to take those parameters into use even though those parameters are wanted by few of the applications. This reset is required to create the first start up scenario when all applications are communicated with the new set of configuration information. Common database shares the concept of “data centricity”, which is defined in the concept of parallel processing. In parallel processing, the task that is represented as data or set of actions is divided into multiple simultaneously operable processing components.

In data centric design, the modified parameters are updated to the database by the master application and the concerned application receives them by subscribing for those parameters. A single BTS system component can be reset without resetting other system components. Thus,

the BTS downtime can be reduced considerably. Assume a network having 1000 BTS and each BTS recovers from some fault at some point of time. In each BTS life time, if a fault recovery takes place in approximately 5 minutes, the network provider loses approximately 5000 minutes in the BTS recovery. With the data centric design approach, the non-operation time of BTS is envisioned to be minimized to a considerable level.

The functionality of a common database will be divided into database wrappers (DBW): DBW Server and DBW Clients. There will be one DBW Server, while several DBW Clients will be running in different processors. From the functional point of view, the DBW Server will be the one of the first applications that starts in BTS SW. The DBW Clients will be part of every application that requires database access. APPENDIX 2 presents the detailed design approach of the common database model.

5.2 Runtime Refactoring

Development time refactoring gave a better look to the project and the original design of the SW became visible to a desirable extent. Development time refactoring does not change the behavioral part of the software, only the faded integrity of the SW is restored by development time refactoring work. Since the runtime refactoring can change the behavioral aspect of the software, runtime refactoring is envisioned to make the BTS O&M SW more robust, asynchronous and suitable for Linux or similar symmetric multiprocessor architectures and concurrently suitable for the existing OSE.

Runtime refactoring is pictured to address the synchronization and scheduling problems to achieve a higher performance of the programs. This leads to the reduction of context switching between the threads. Context switching occurs in a thread based kernel system. The BTS O&M SW is designed to function in a parallel processing environment with the help of Rhapsody threads. Smaller thread amount puts less overhead to OS for context switching in runtime.

5.2.1 Runtime thread reduction

To achieve parallelism, the BTS O&M SW is designed to run as parallel Rhapsody threads. Rhapsody provides an easy way to create parallel threads. Parallel threads communicate with each other in runtime in order to continue with different functionalities. During the runtime, the operating system has to provide a harmonious behaviour between the threads by running them in a suitable execution order. The kernel does the context switching from executing one thread to executing another. The kernel saves the context of the currently running thread and resumes the context of the next thread in line. In a thread-based kernel, the kernel manages context switches between kernel threads, rather than processes. Context switching occurs when the kernel switches from executing one thread to executing another. The kernel saves the context of the currently running thread and resumes the context of the next thread that is scheduled to run. The context switching is done when:

- The time slice of the thread expires and a trap is generated or
- A thread puts itself to sleep, while awaiting a resource or
- A thread puts itself into a debug or stop state or
- A thread returns to the user mode from a system call or trap or

- A higher-priority thread becomes ready to run.

The context switching generally requires an intensive computation and a considerable processor time. Thus, context switching eats up CPU time. However, this has not been easy to repress in practice. The major focus on the designing of operating systems has been to avoid unnecessary context switching. Linux claims to be an operating system that has an extremely low cost of context switching and mode switching. Although the future of the BTS O&M SW is seen to be running on top of Linux or a similar symmetric multiprocessor architecture it is still good to minimize context switching between the running threads.

Thus, In order to improve the stability and performance of O&M SW in the Linux operating system, the thread reduction routine has been designed. Deterministic behavior of O&M SW was envisioned by decreasing the number of threads, which would simplify the software process architecture. A similar approach of thread reduction was not designed for OSE based O&M SW due to the limitation of stack usage per OSE process.

The main class of a subsystem with a state chart is set to be active, meaning that active object is created in the runtime. The behavioral part of the class initializes and starts executing in the active thread when the startBehavior() call is made on that instance of the class (see Section 2.4). A user defined thread name is given to recognise the thread and is useful in debugging if the kernel raises any error. The rest of the reactive classes (classes with state chart) in that subsystem are assigned to run the primary thread. Each of the concurrently active class behavior is set to run on its own thread.

The thread reduction work required that the Rhapsody control of creating threads is changed to manual thread creation. An xml file with the existing thread info is created where the set of non-semaphore threads are combined as the alias of a super thread. The resource hungry threads are kept independent in the xml file. APPENDIX 3 illustrates the thread info in an xml file.

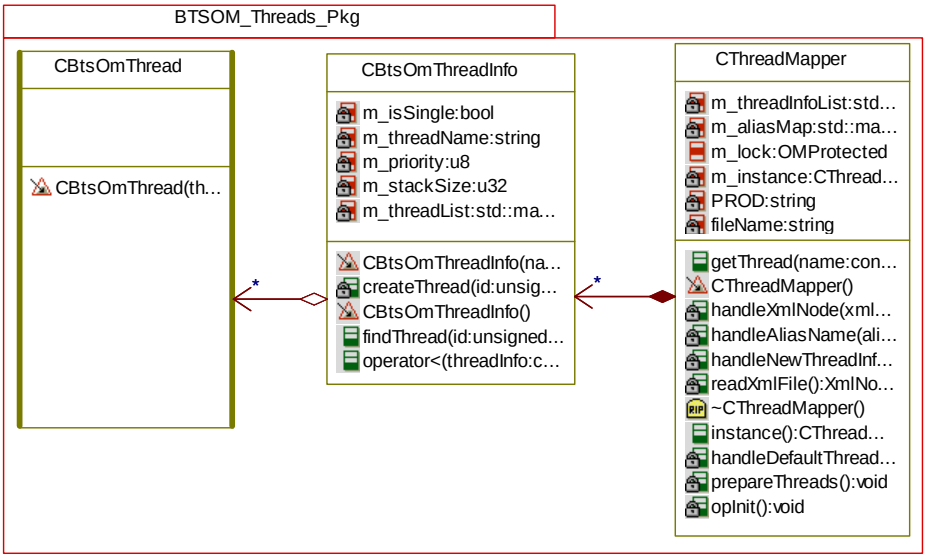

Figure 5-7 Thread reduction object model diagram

A thread-mapper class is designed to provide the thread information from the XML file. Figure 5-7 illustrates the Object Model of the design. CThreadMapper is the interface towards the other subsystem classes and provides thread information read by the CBtsOmThreadInfo class from the xml file. CBtsOmThread activates the thread to start the event-processing loops.

All active classes are set to be sequential in the Rhapsody model. When an older active object (see Section 2.4) is created, it gets the thread info from the thread-info class and the thread is mapped using the thread-mapper. Thus, active threads are created manually and a few of the active threads are merged to run on a set of the super thread. Figure 5-8 shows the sequence diagram of the thread mapping process.

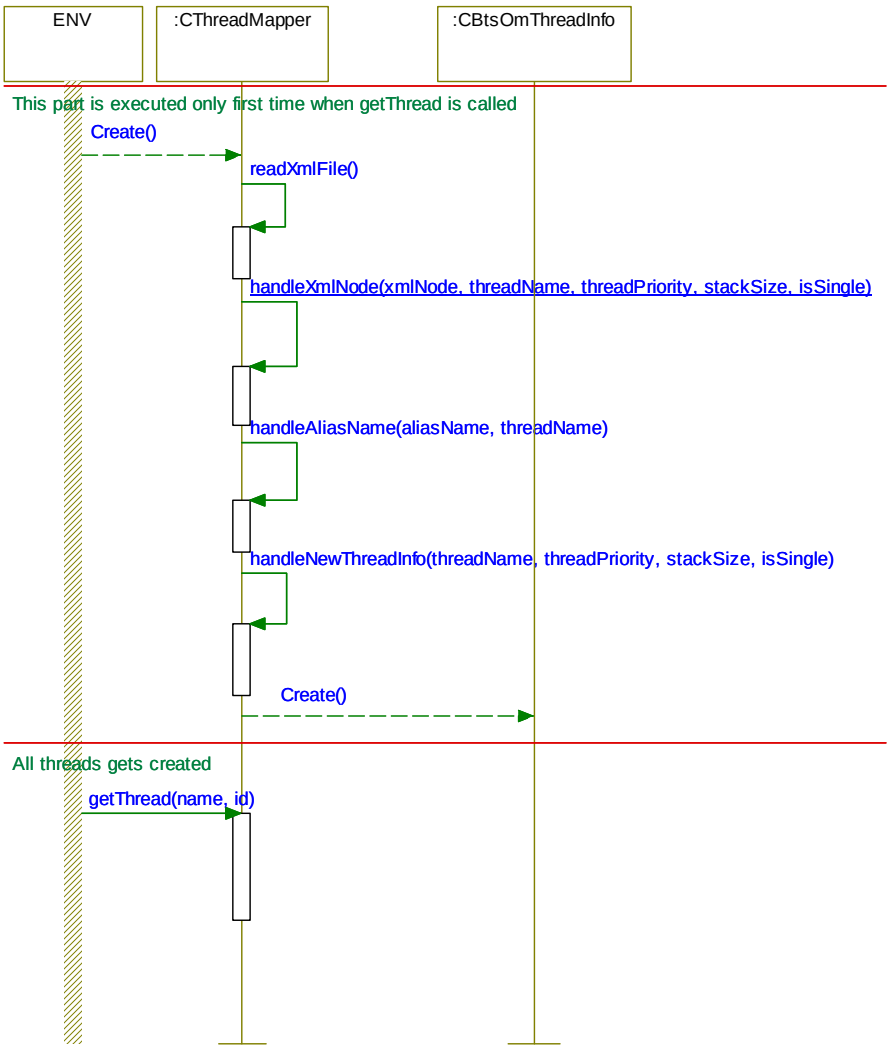

Figure 5-8 Thread reduction sequence diagram

In the Linux version of the software there were about 136 Rhapsody threads before this modification, which was reduced to the number of 84 threads.

5.2.2 Un-controlled thread priority

The first thread of a Rhapsody based multithreaded application is called the mainThread. The mainThread is also the system thread, and objects that have sequential concurrency run on this thread (Rhapsody Help, 2010). During the thread reduction activity it is found out that several sequential objects are actually mapped to the system thread. In other words, those sequential threads are not bound to the supervisor or master thread. Thus, the child thread runs on a

different or altered priority, either lower or higher than the supervisor thread. This alter priority upsets the synchronization of the execution of the threads. The child threads either run with a higher or lower priority in comparison to the supervisor threads. Thus, OSE based O&M suffers random scheduling of the execution of the master and child threads.

Such alter priority problems are fixed by binding the child threads to the master or supervisor threads and thus keeping the static priority of the threads in control.

5.2.3 Unsynchronized start up behavior

Reduction of threads has made O&M SW more deterministic. However synchronization between threads has still been required to achieve the presumed stability of the SW. During the thread minimizing activity, when threads are created manually, the software has started behaving erratically. The situation has been as if the synchronization between some of the running threads has been completely lost. Events are getting lost in transition. So, an exercise to synchronize the start up behavior has been taken into account. The following sections describe the problems and provided solutions.

5.2.3.1 Inappropriate startBehavior() call

It is found that the startBehavior() call for several active classes is not correctly executed. When a constructor is called, the class instance gets created and the startBehavior() call must be executed on the active or reactive instance. The behavioral part of a subsystem must immediately be ready after the instance is created; otherwise it may miss the reception of an event sent by other subsystems. (See Section 2.4 for the definition and usage of start behavior)

All startBehavior() calls are checked and corrected in a way that the method is called when the instance is ready to initialize the behavior of the statechart.

5.2.3.2 Untimely event subscription

BTS O&M SW is designed in a way that a subsystem instance can subscribe for certain event. This is similar to a client-server methodology where a server broadcasts a signal and the

interested clients receive the signal. When a subsystem publishes or broadcasts an event, the subscriber receives the event to continue its behavioral functionality. It is observed that event subscription is done even before the behavior of the state chart has been started. Thus, the subscribed event is lost. And, if the subscribed event is execution critical, the thread waits indefinitely for the subscribed event. All such subscriptions are moved to the state chart where the statechart is ready to receive the subscribed event.

5.2.3.3 Unsynchronized distribution framework usage

BTS O&M is a distributed system where communications between the processes residing in different processors take place. The distribution framework is used to send and receive such communication messages between distributed processes. The basic idea is to get the distribution framework ready for a class instance before it can start communicating. In O&M code there has been several of such cases where the distribution framework for the class instance is not ready when the behavioral part of code had started to execute. In such cases there is a possibility to lose the distributed event or message.

Thus, corrections are done for such code in such way that the distribution framework is ready to serve the class instance before the instance starts sending and receiving messages.

5.2.4 Service based O&M SW

BTS O&M SW is communicative in nature, which means that every subsystem process holds dialogues with other processes in order to continue with the start up and runtime execution. These dialogues between the subsystems follow the client-server communication mechanism. A server provides a service and a client consumes the service. In BTS O&M this client-server relation is not always unidirectional. The client-server relation changes depending on the priority of the service that executes functional behavior of the system.

Thus, to make O&M a service-base system, the services must register themselves into a repository from which a consumer subscribes for the required service(s). As and when a service is ready, the subscriber gets this info from the repository. This is called ServiceRegistry and it is an important tool to synchronize O&M SW. O&M SW already posses a request/reply/notification

mechanism in place where there are handshaking dialogues between the service providers and subscribers. ServiceRegistry is a replacement for the existing polling style.

ServiceRegustry is designed in the way that a service provider registers its service as local or global. A service is said to be local when the service is not extended towards other nodes, and a global or public service is one available across the nodes. Local services are distributed to the subscribers when a service is registered. Global service, on the other hand, is first distributed between the ServiceRegistry instances running in different nodes, and then the ServiceRegistry in the other node distributes the service towards the local subscribers. Thus, all subscribers are aware of the services when they are ready to serve. Once the server is ready, the clients can start communication towards the server, and the system continues with its behavioral functionality. The server withdraws the service from the ServiceRegistry when it no longer wants to serve. Service registration and subscription is designed to be synchronous (method-based) while “server ready” and “server stop” info are asynchronous (event-based). Figure 5-9 shows a simplified single node ServiceRegistry sequence diagram.

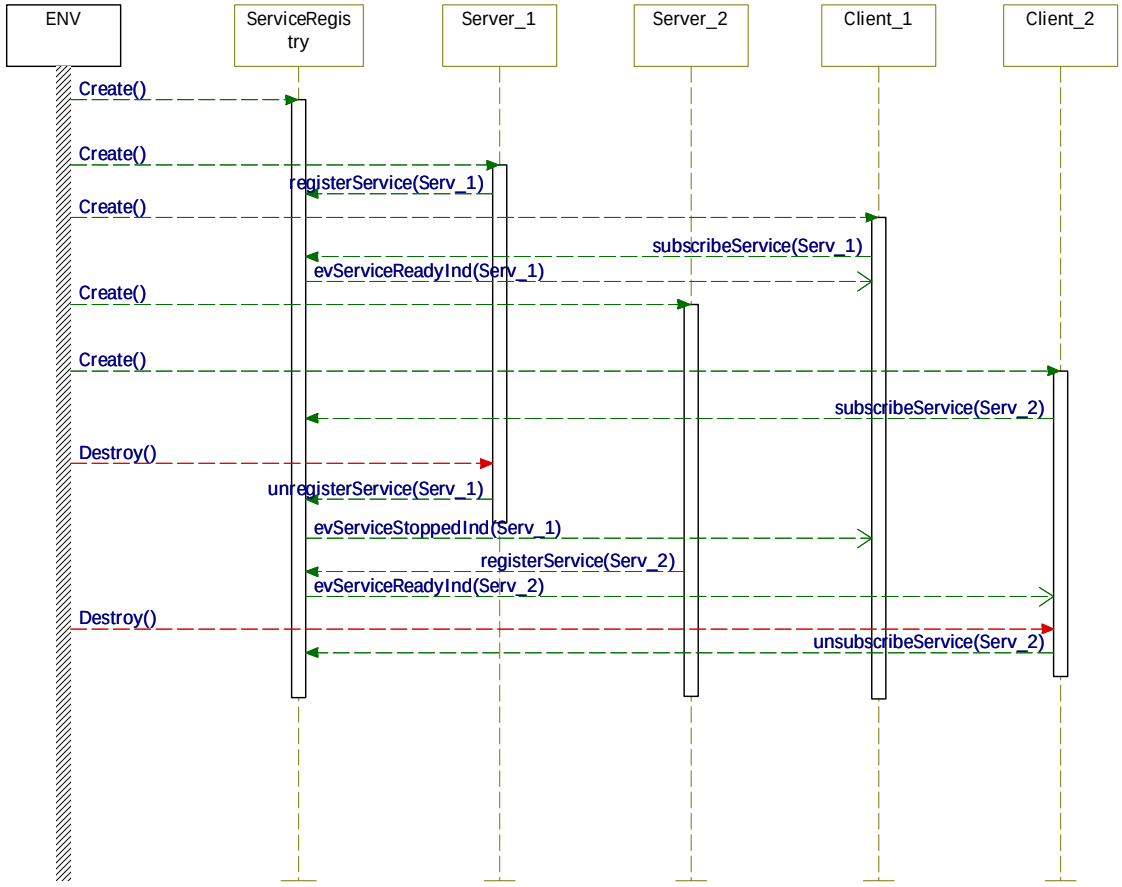

Figure 5-9 Simplified ServiceRegistry sequence diagram

5.2.5 Asynchronous semaphore

BTS O&M uses parallel state charts to specify the parallel behavior of the subsystems. Those parallel state charts actually run on the same process or thread context. Thus, if any of the parallel state waits for a blocking service, for example a semaphore to be released, it blocks the whole process even if there is a possibility for the parallel state chart to continue. In other words, synchronous blocking affects if the process or threads are running in the same context. During the thread reduction activity threads are combined to a super thread (Section 5.2.1). If any of the alias threads gets blocked, the whole context is blocked. Such a problem can be overridden by using the concept of asynchronous semaphore. Figure 5-10 illustrates a synchronous semaphore condition. In this example it is assumed that a second parallel state chart can continue with its behavior. When the method opStartDoingSomething() calls for a synchronous semaphore lock, the second state chart awaits the semaphore lock to be released because the parallel state chart is executed in the same context as that of the first state chart.

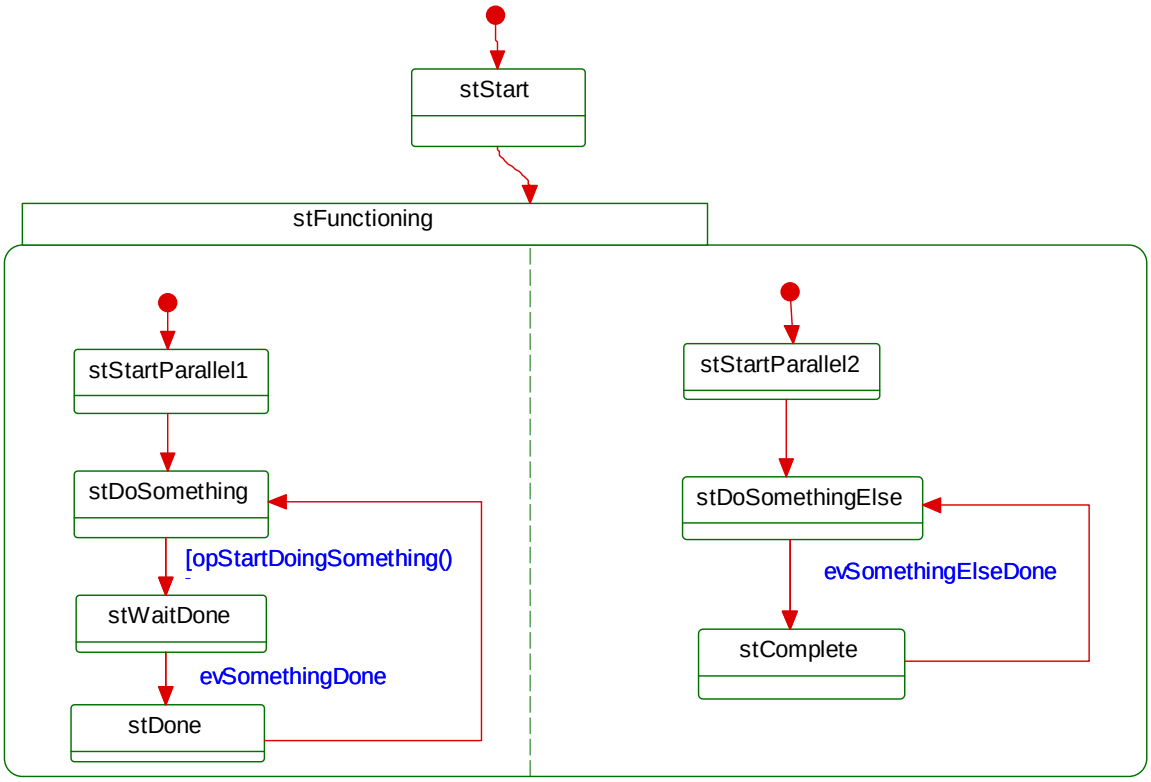

Figure 5-10 Parallel state chart with synchronous semaphore

This problem of the synchronous semaphore can be overcome with asynchronous messaging. Whenever there is a need for mutual exclusion, the synchronous interface might be changed to the asynchronous interface. In the above example, this is done by replacing the method call in the first state chart by an asynchronous UML event trigger. This gives the parallel state charts a chance to run independently of each other. Of course, if the semaphore is required to restrict the process or thread as a whole, the semaphore should be acquired via the synchronous interface.

A separate asynchronous class can be made that keeps track of the semaphore to be locked and released. See Figure 5-11 for the modified state chart. In this example, stDoSomething state sends the evWaitSemaphore event to the tracking class when the semaphore has to be locked and sends the evReleaseSemaphore event when the lock is no longer required.

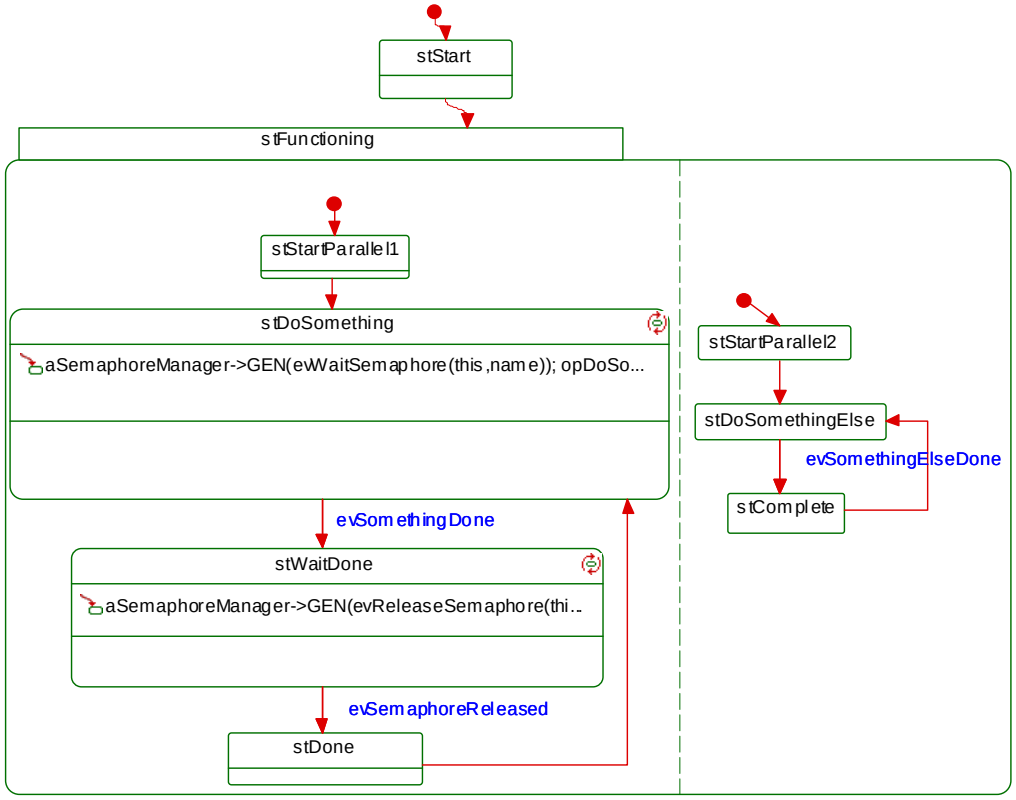

Figure 5-11 Parallel state chart with asynchronous semaphore

5.2.6 Mixing timeout and timer

Rhapsody provides UML based state charts to design the behavioral part of the code (see Section 2.4). In the statecharts, timeout is used to continue to the next state if the expected event is not received. This generally leads towards an error or a failure in the functional behavior of the system and sometimes calls for recovery. It has been found out that the timers are used as timeouts in state charts of O&M SW.

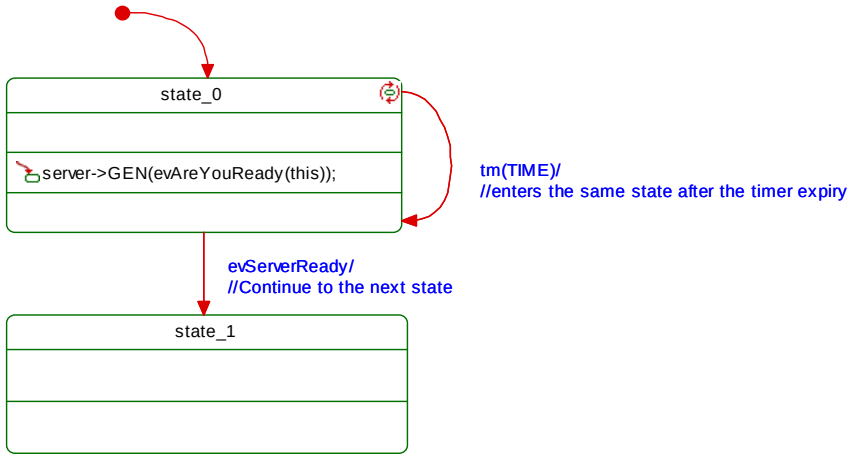

Figure 5-12 Example: Mixing timer and timeout

Figure 5-12 shows an example of such an implementation. In this example, the client asks for readiness of the server and keeps on asking the server until it replies. In such cases, ServiceRegistry (Section 5.2.4) must be used. The client should subscribe for the service from ServiceRegistry, and waits until it receives the registration info from the ServiceRegistry (Section 5.2.4).

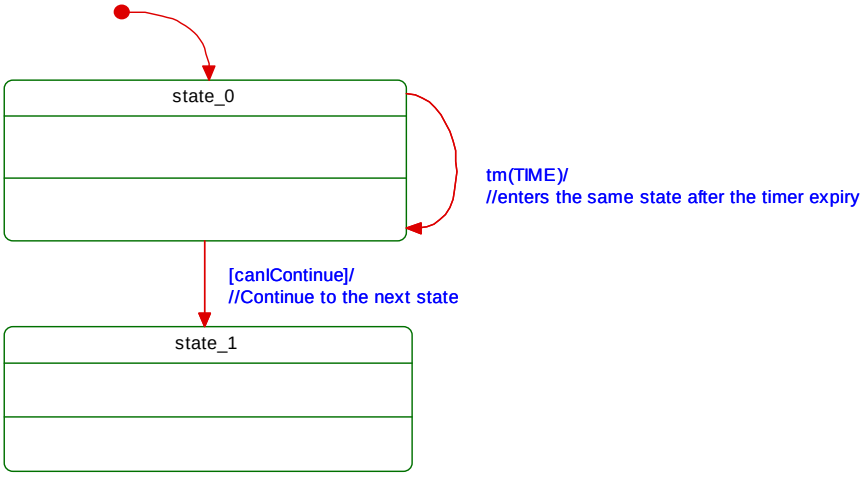

Figure 5-13 Example: Inefficient polling

Another example is in Figure 5-13 where the state continues to state_1 when the canIContinue condition is true. Such implementations should be replaced with either the ServiceRegistry-based client-server style (Section 5.2.4) or simply wait for the required event to be received.

5.2.7 Usage of timer as conditional transition

BTS O&M is an event driven system where the functional behavior of a state chart proceeds with the reception of an event. Sometimes this basic principle of O&M design is violated by a wrong usage of a timer.

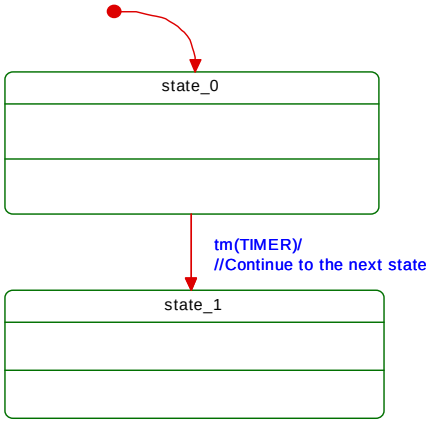

Figure 5-14 Example: Misusage of timer as condition

Figure 5-14 shows such an example. In this example the state transition from state_0 to state_1 happens only after the expiry of the timer TIMER. In such places, the client requests for a service and waits for fixed time (timer) for the action to be ready. Such a timer does not guarantee that the service is ready and the requested service is not synchronized with the action in the client. If the execution environment of the software changes, such a timer may also need to be changed, and is thus surely inefficient. The ServiceRegistry concept discussed in Section 5.2.4 is a recommended solution to this problem.

Such a waiting might still be necessary in the exception path of the statechart where it is ensured that while the server is never ready with the requested service, the client can still proceed and either inquires for an alternative or stops executing if the requested service is critical for the client operation. For an obvious reason, such an exceptional timer is required to be calculated based on the best worst-case scenario.

5.2.8 Unprotected re-entrant

In Rhapsody, a protected object is an object with at least one protected operation. Operations that are protected are called guarded operations. In a multiprocessor environment, it is quite natural to protect or guard the critical part of the code or complete method to make it re-entrant. The Rhapsody OXF framework provides a simpler way to protect the parts of the code by using the OXF framework. In the previous design, this protection has not been safe for the multiprocessor environment. In the following code example the getInstance() method is designed to be re-entrant and therefore the instance is created only once. A static variable "..will be initialized only the first time thread execution reaches its definition" (Stroustrup, 2004). In the mentioned example 1, the static variable protect is initialized when the method is entered. However, in a multi-core or multiprocessor environment there is a possibility that two threads enter the method at the same time resulting in the protect variable to be instantiated twice. In a singleton case, the first call to the method does not yield the expected result causing the protection to fail.

Example code 1:

```
class mySingletonClass {
        static OMReactive* instance = null;
        public:
        static OMReactive* getInstance()
        {
```

```
                Static OMProtectd protect;
                OMGuard guard(protect); // guard the instance creation.
                if(instance == null)
                        Instance = new mySingletonClass();
                return instance;
        }
}
```

Note (Rhapsody Help, 2010):
OMProtected is the base class for protected objects.
OMGuard is used to make user operations guarded or locked between entry and exit

Such code should be changed in a way that the static variable protect is created already before the class is instantiated. In the following code the static variable protect is set as the class attribute and is instantiated well before the getInstance() method is called. Thus, the method is guarded safely even for a multiprocessor environment.

Example code 2:

```
class mySingletonClass {
        static OMReactive* instance = null;
        static OMProtected protect;
        public:
        static OMReactive* getInstance()
        {
                OMGuard guard(protect); // guard the instance creation
                if(instance == null)
                        instance = new mySingletonClass();
                return instance;
        }
}
```

Example code 2 has redundant guarding. Actually, the idea is to create one instance of the singleton class. If the getInstance() method is called n times during the course of the program, the locking or guarding is not necessary for (n-1) times. This inefficiency might be removed by double checking if guarding or locking is needed. In example code 3, this double checking is done in a way that should be followed in BTS O&M when a singleton class is initialized.

Example code 3:

```
class mySingletonClass {
        static OMReactive* instance = null;
        static OMProtected protect;
        public:
        static OMReactive* getInstance()
        {
                If(instance == null) //check if the instance is uninitialized
                {
```

```
                OMGuard guard(protect); // guard the instance creation
                if(instance == null)
                    instance = new mySingletonClass();
            }
            return instance;
        }
}
```

In example code 3, the protection is done only once even if the getInstance() is called several times during the course of the program. It is also to be noted that guarding a critical code part must be unique. If there are multiple methods to access the same protected data, all these methods must be protected using the same guard.

5.2.9 Performance vs. runtime binary size

The GNU BTS O&M compiler (GNU GCC) offer options for squeezing more performance out of the application code. Many of these optimizations, however, tend to increase the overall code size. It is recommended to adapt an iterative process of building application code by using different compiler optimization options and profiling the result. This way infrequently used and non-critical section of code can be identified where it matters least for the reduced code size to trade off the performance, resulting in the minimum impact on O&M SW. A critical computational loop performance can be increased, resulting in a better overall performance of the system (Li, 2005). In BTS O&M initialization phase, increasing performance of such critical loops would decrease the overall start up time of BTS.

The following code examples show different implementations of the for-loops with the same output:

Example code 1. (Li, 2005)

```
for (unsigned integer i = 0;  i < 12; i = i + 1)
{
        sum = coefficient * data;
        coefficient = coefficient + 1;
        data = data + 1;
}
```

Example code 2.

```
for (unsigned integer i = 0;  i < 12; i = i + 2)
{
        sum = coefficient * data + (coefficient + 1) * (data + 1);
```

```
        coefficient = coefficient + 2;
        data = data + 2;
}
```

Example code 3.

```
for (unsigned integer i = 0;  i < 12; i = i + 4)
{
        sum = coefficient * data + (coefficient + 1) * (data + 1) + (coefficient + 2) *
                                                    (data + 2) + (coefficient + 3) * (data + 3);
        coefficient = coefficient + 4;
        data = data + 4;
}
```

Example code 1 provides the smallest code size at the cost of the least performance, while example code 3 provides the highest performance at the cost of the largest code size. Example code 2 lies between these extremes. Thus, to decrease the start up time of the BTS, such loops, if executed in start up, can be unrolled (as in example code 3), and the method, containing the for-loop, can be in-lined. Although this increases the code size, a better performance decreases the start up time.

There are few examples in the existing O&M design to decide the execution of code in runtime. Sometimes, the code is separated between the products by runtime variability; for example,

```
if (isProductA())
{
        //execute product A specific code
}
else
{
        //execute other product specific code
}
```

In runtime, the method isProductA() returns a value that makes sure that the part of the code is run only for the intended product (A). Such code should be removed to decrease the binary of the executable. This would free up some memory that can invariably be used by other parts of the code where the performance might be achieved by increasing the code size.

6 FURTHER RESEARCH AND STUDY

The completed implementations and recommended solution presented in this thesis is only the beginning of a voyage. Refactoring is a continuous process and synchronization problems are very common in a multithreaded environment. The discussed solutions in this thesis are not exhaustive; there will be problems popping up and those have to be addressed. What could potentially be a problem in the future is the synchronization problem between different HW having their own processors running the same O&M SW. Different sets of HW might be running at different CPU speeds and would need to be addressed separately. The ServiceRegistry concept might be a savior in this case.

ServiceRegistry is designed to provide the service provider address. However, the physical accessibility towards other service providers may also be direct. The ServiceRegistry service may be limited to acquire server-client synchronization. This means that client simply has to know when the service is ready and uses the hard-physical address pointers to communicate.

The O&M internal ServiceRegistry concept may be extended to the whole BTS system using a common SW application. A common SW application is a kind of a middleware between the applications running in BTS. Thus, naturally, if the concept has to be extended for all applications, the registrar should be residing in a common place. Nevertheless, O&M internal ServiceRegistry still plays its role for O&M and a new interface may be created between O&M and Common SW ServiceRegistry.

The common Database design is not conceptually proved, but has the potential to reduce the start up time by moving the extra overhead of providing data from O&M to a standalone application. Common Database complements the parallel processing and may provide a mechanism to restrict the problems or failure of a BTS application to a smaller area of operation resulting in a faster recovery from the failure. Ideally, all applications in BTS SW should work simultaneously to achieve the common objective of reduced starting up time of the BTS and keep the BTS running without any major fault in operation. A common database containing the configuration information and the user defined properties gives asynchronous and synchronous accessibility to the database to be used in the start up and runtime. Thus, the O&M SW application does not need to act as a data provider for other applications and it only controls the BTS functional operations.

Thread reduction activity is mainly carried out for the Linux based O&M SW, while OSE based O&M still has the same amount of threads. The good point is that alter thread problems are solved during this task. This has given OSE based O&M a similar structure for thread creation and control. Since the OSE based environment has limited stack space for each thread, combining the threads is not a simple task. Nevertheless, the good news is that due to the external control of the thread creation, few of the possibilities may be tried. Combining the threads with a smaller stack size might be one of such possibilities. In addition, changing the synchronous interface for block or semaphore to the asynchronous interface gives the possibility to combine even more threads to a super thread.

The problems and solutions provided in this thesis are not limited to BTS Operability SW. It is applicable to all message based systems. Client-Server concept, multithreaded applications with the minimum threads or processes, achieving a high performance with smarter coding, achieving reliability with synchronous behavior of the process or threads, restricting problems in a smaller area of operation, are few of the many to be highly recommended for any SW system. The problems in the multiprocessor architecture have been widely discussed and there are materials available which give hints for a better architectural design. It is the design architecture of the system that gives a solid ground for the efficient and less error prone software. When every little detail such as interface, accessibility, coding guidelines is addressed in the architecture of a system, the software design may achieve the expected standard.

The quantitative measurement, for example in number of code lines, does not suit the need of the refactoring project. The provided solutions and recommendations are qualitative in nature. BTS start up time is not decreased significantly, however an asynchronous system is made more deterministic by providing synchronization between the subsystems. All techniques presented in this thesis are to achieve a better O&M application that is reliable, robust and free from the scheduling policies of OS. The performance problems are covered with the same zest as those with the synchronization. Few of these techniques simply have to be proved before they can be strongly advocated.

7 CONCLUSION

This thesis was a vast learning experience. From the perspective of the BTS O&M SW, each of the runtime refactoring concepts was fresh and more stimulating than the other. Each of these concepts and findings were intended to achieve an OS independent, robust and reliable O&M SW. The methodology of the thesis was to first prove the concept by trying several styles and design approaches. Once the best of the results were achieved, the concept was set to release. Among all the runtime refactoring concepts, the followings were implemented during the thesis work:

- Runtime thread reduction
- Controlled thread priority
- Synchronized start up behavior, i.e. proper startBehavior() execution, timely event reception and synchronous distribution framework usage
- Service based O&M SW
- Protected reentrant

The work carried out during this thesis is not free from criticism. There had been many discussions and re-discussions, and designs had been reviewed; some concepts were postponed due to lack of time and needs of the situation. The BTS O&M had been a live project, and most important was to get the SW released in a strict schedule. There had been a few obstacles faced during the thread reduction phase, where skepticism was high and the project achievement was viewed to be less. Even so, all these obstacles were overridden when the result was soon proved to be beneficial. Thread reduction stabilized the O&M SW in the Linux operating system, and simultaneously OSE based O&M SW became more controlled.

Thread reduction for Linux and ServiceRegistry were the headliners of this work. Solving other potential ambiguities during this work had been added advantages. Several runtime synchronization problems between the processes were solved during the work. The basic synchronization problems in Linux were addressed and a solid mechanism was provided to continue with the future developments for both Linux and OSE and other similar symmetric multiprocessor architecture.

This thesis gives a guided platform for the future O&M development projects. The BTS O&M SW is an ever growing application and enormously complex. Such a large complex system is developed by people of various knowledge bases and divided by location and time. This thesis

gives the developers few tested and few untested techniques. Event though the achievement is not measurable in numbers, a significant quality improvement is easily visible. This gives an added confidence towards O&M software that is alive in BTS for a decade and will continue to live with spectacular pride and prejudice.

During this thesis, the main objective of reducing the O&M threads has been achieved for Linux, and the possibility of reducing the threads in OSE has been discussed. The threads are synchronized to run in harmony in both the Linux and OSE based architecture. Finally, few innovative techniques, such as synchronized distribution framework, server-client based architecture and controlled thread creation are provided to make the future development work safer and more reliable. Development time refactoring emphasized the alignment of the BTS O&M SW into the original architectural model by introducing the defined interfaces and controlled association between the subsystems.

REFERENCE

Aeolean Inc. (2002, December 11). *Introduction to Linux for Real-Time control.* Retrieved April 2011, from http://www.aeolean.com: http://www.aeolean.com/html/RealTimeLinux/RealTimeLinuxReport-2.0.0.pdf

Arjona, J. L. (1999). *Design Pattern for Communication Components of Parallel Processing.* Retrieved from http://www.matematicas.unam.mx: http://www.matematicas.unam.mx/jloa/publicaciones/multipleRemoteCall2.pdf

Bovet, D. P., & Cesati, M. (2005). *Understanding the Linux Kernel.* O'Reilly.

Cucinotta, T., Giani, D., Faggioli, D., & Checconi, F. (2010). *Effective Real-Time Computing on Linux.* Retrieved from https://www.osadl.org: https://www.osadl.org/fileadmin/dam/rtlws/12/Giani.pdf

Edmund Mayer. (2005). *Using Rhapsody in C++ with Your Middleware.* Retrieved April 2011, from http://www.swd.ru: http://www.swd.ru/files/share/Rhapsody/materials/Whitepapers/Using_Rhapsody_C_with_Middleware_Whitepaper.pdf

ENEA. (2011). *Enea OSE: Multicore Real-Time Operating System (RTOS)*. Retrieved from www.enea.com: http://www.enea.com/Templates/Product____27035.aspx

ENEA. (2000). OSE Real –Time Kernel, Manual. ENEA OSE Systems AB.

ENEA. (2011). *OSE Realtime Operating System (RTOS)*. Retrieved April 2011, from www.enea.com: http://www.enea.com/templates/Extension____12765.aspx

ENEA. (2008). *The Architectural Advantages of Enea OSE in Telecom Applications*. Retrieved April 2011, from www.enea.com: http://www.enea.com/epibrowser/Literature%20%28pdf%29/Pdf/Leadgenerating/White%20papers/Enea%20WP%20Advantages%20of%20OSE%20in%20Telecom%20Apps.pdf

Flynn, M. J. (1972, 2009). Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers , C-21* (9).

Fowler, M. (1999). Refactoring: Improving the design of existing code. In M. Fowler, *Refactoring: Improving the design of existing code* (pp. 53-74). Addison Wesley Longman Inc.

HP White Paper. (1997, April 7). *HP-UX Process Management.* Retrieved April 2011, from http://docs.hp.com/en/5965-4642/5965-4642.pdf

IBM rational Rhapsody. (n.d.). Retrieved April 2011, from http://wn.com/: http://wn.com/IBM_Rational_Rhapsody

Jones, T. M. (2007, March 14). Linux and symmetric multiprocessing:Unlock the power of Linux on SMP systems. IBM.

Jussi Leppanen. (2010). BTS O&M Architecture Specification. OULU, Finland: NSN Internal Document.

Jussi Leppanen. (2010). BTS O&M Process Architecture. OULU, Finland: NSN Internal Document.

Li, R. (2005, November). *How to Reduce Code Size (and Memory Cost) Without Sacrificing Performance.* Retrieved April 2011, from http://embedded-systems.com: http://embedded-systems.com/design/174402683

Linux Manual. (n.d.). Linux man pages.

Maurice Herlihy, N. S. (2008). *The Art of Multiprocessor Programming.* Morgan Kaupmann Publishers.

Meyers, S., & Alexandrescu, A. (n.d.). *C++ and the Perils of Double-Checked Locking.* Retrieved April 2011, from http://www.aristeia.com: http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf

Rhapsody Help. (2010). *IBM Rational Rhapsody*. Retrieved April 2011, from http://publib.boulder.ibm.com: http://publib.boulder.ibm.com/infocenter/rhaphlp/v7r5/index.jsp

Stroustrup, B. (2004). The C++ Programming Language. Book by B. Stroustrup, *The C++ Programming Language* (s. 145). Addison Wesley Longman Inc.

Thornley, J. (1997, October 7). *Multiprocessing Architecture.* Retrieved April 2011, from www.cs.caltech.edu/: www.cs.caltech.edu/~cs284/lectures/7oct97.ppt

APPENDIX 1 PROCESS AND THREAD

- Linux Process: A Linux process is a program in execution.
- Linux Thread: A Linux thread is a context of execution, which is scheduled by a Linux kernel. Linux threads shares the memory space. In Linux based O&M, Rhapsody threads are mapped into Linux threads.
- OSE Process: OSE process is the context of execution, which is scheduled by OSE. OSE processes share the memory space. In OSE-based O&M, Rhapsody threads are mapped into OSE processes.
- Rhapsody Thread: Execution context controlled by the Rhapsody OXF-Framework. It is invisible to the operating system. OXF schedules the threads in the FIFO-manner from its thread list (Leppanen, 2010).

APPENDIX 2 COMMON DATABASE DESIGN APPROACH

DB Wrapper

The common database is accessed by the database wrapper (DBW). The DBW is divided into client and server. The server part of the wrapper is linked to the database and interprets the clients' data requests. The DBW server also hides the database from the client/application. During the runtime, the server also updates the database as commanded by the master application. The client, on the other hand, serves the applications and hides the interface and communication mechanism with the server. The interface between the application and client is method based. During compile time, the application is linked with the client. During the start up process, the server is created by O&M SW, and the clients are created by the applications.

Common database

Like all the databases, this common database consists of an organized collection of data for multiple uses by different BTS applications. The database is a collection of C++ classes. During the BTS start up, the master application creates the database by using the DBW server application. The DBW server initializes the data classes and keeps a list of objects for easy searching. The common database is expandable and is independent of interfaces provided by the DBW server.

Master application

BTS O&M SW is the master application. The master application and the DBW server run in the same memory space. During the start up of the BTS, O&M SW creates the DBW server. When the O&M SW interprets the configuration file, it starts commanding the server to initialize the data classes, one by one. During the runtime, if any configuration information changes, the O&M SW commands the server to update the corresponding data class object.

Database Accessibility

The O&M SW creates the database during the start up. During the runtime, even O&M SW cannot access the database directly. The full access of the database lies with the server, while others can request the server for the required set of data. Thus, all applications can access the database via the server. To control the database, it is necessary to provide a write access to the

chosen few applications. This decision depends on the functional requirement of the system. In cases where O&M SW does not have to be involved for certain data, one application can have a write access to the database and another application reads that data in the runtime. Another possibility is to give write access to O&M SW only. Any application which has to update database, can request O&M SW. Although this design protects the data, it still does not fulfill the basic idea of data centricity. O&M SW has to carry this extra burden of updating the database even for the other applications.

Interfaces

MASTER APPLICATION – DBW SERVER: This interface is method based. The purpose of this interface is to provide services for the DBW server creation and the creation and runtime modification of the database.

DBW SERVER – DATABASE: Method based interface. The DBW Server keeps a list of the data class objects. During search operation, the list is checked for the existence of the invoked data.

DBW SERVER – DBW CLIENT: This is the most important interface that hides the communication mechanism between the server and client parts of the database wrapper. The interface is both method and message based. When the server and client applications are running as a single executable, the interface provides a method based communication mechanism; and message based communication is provided when the client and server are running as separate executables. The data part of the message contains the required commands as specified below. The data part of the message is constructed using a pre-defined format [RAML or XML].

DBW CLIENT – APPLICATION: This interface is method based. The application and client may also need a middleware to overcome the development language barrier.

Development methodology

The DBW application is developed using C++. The application is divided into two parts. The DBW server and client work as communication component, command interpreter and most importantly database wrapper. The client hides the source of the data from the application and thus works as a rendezvous. There is one instance of the DBW server be available in runtime, where as several

DBW clients communicate with the server. A BTS application that requires data to be fetched or set is linked to a client application during the compile time. Compile time linking may be replaced by dynamic linking at runtime.

Functionality of DBW Server

From the functional point of view, this application is the one of the initial applications that starts in BTS SW.

• Server is created by a master application • Server starts as a common software process and registers itself for receiving system communication messages • Server receives client registration and stores the registered client • Based on the command from master, server populates data and store a list of populated data • Once the data population is done in start up, server informs the registered clients about the data readiness and flushes out client registration info • Server waits for client request for data • Server receives data request message and parses the request for fetch, set or subscription • Server searches for data requested • Generates the message with the searched data, and finally • Server returns the data in a message or method call

Functionality of DBW Clients

DBW clients are created when a BTS software application comes into existence. The client count may be equal to the number of applications running in BTS.

• Application create the DBW client • Client inquires the server address from common software application • Client generates registration message and sends to the known server address • Client stores the node info to invoke correct interface methodology: message or method based • Client waits for readiness message from server and provides this information to the application • Application request for data • Client parses the request and makes system communication message and sends to server or calls appropriate method in server. • Client receives the requested data and parses the information as required by the application and returns

Command Set

Fetch: This request is to get the data from common database.

Set: This request is to initialize and modify the existing data

Subscription: An application may subscribe the data parameters for a change from the server. This is required in such cases where one application changes the data parameter value and another one requires the changed value. Server stores the subscription sends data change notification.

Un-subscription: If an application does not require the changed data anymore, the data parameter is un-subscribed from the server. This prevents the server to send an unnecessary data change notification to the clients.

Registration: The DBW client application registers itself with the DBW server to receive the database readiness indication. After readiness indication, the DBW server is ready to provide data to the client.

Un-registration: The DBW Client application un-registers itself from the DBW Server if the application dies in controlled manner.

APPENDIX 3 EXAMPLE CONTENT OF XML FILE FOR THREAD MINIMIZING ROUTINE

```
<?xml version="1.0" encoding="UTF-8" ?>
<threads>
<thread name="Rhaps_IndependentThread1">
        <priority>12</priority>
        < stackSize >10240</stackSize> <!—multiple of 512 -->
        <single>1</single>
</thread>
<thread name="Rhaps_IndependentThread2">
        <priority>15</priority>
        < stackSize >5120</stackSize>
        <single>1</single>
</thread>
<thread name="Rhaps_SuperThread1">
  <alias>Rhaps_AliasThread1</alias>
  <alias>Rhaps_AliasThread2</alias>
  <alias>Rhaps_AliasThread3</alias>
  <alias>Rhaps_AliasThread4</alias>
        <priority>16</priority>
        < stackSize >15360</stackSize>
        <single>1</single>
</thread>
<thread name="Rhaps_SuperThread2">
  <alias>Rhaps_AliasThread1</alias>
  <alias>Rhaps_AliasThread2</alias>
        <priority>11</priority>
        < stackSize >15872</stackSize>
        <single>1</single>
</thread>
</threads>
```

The attribute <single> tells the thread constructor class to set the alias thread to the super thread.

This also tells the thread constructor class to set a single context for threads with the same name.